Temas de “Programación funcional” (curso 2012–13)

José A. Alonso Jiménez

Grupo de Lógica Computacional
Dpto. de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla
Sevilla, 26 de Septiembre de 2012

Índice general

Tema 1

Introducción a la programación funcional

1.1. Funciones

Funciones en Haskell

- En **Haskell**, una **función** es una **aplicación** que toma uno o más **argumentos** y devuelve un **valor**.
- En Haskell, las funciones se definen mediante **ecuaciones** formadas por el **nombre de la función**, los **nombres de los argumentos** y el **cuerpo** que especifica cómo se calcula el valor a partir de los argumentos.
- Ejemplo de definición de función en Haskell:

```
doble x = x + x
```

- Ejemplo de evaluación:

	doble 3	
=	3 + 3	[def. de doble]
=	6	[def. de +]

Evaluaciones de funciones en Haskell

- Ejemplo de evaluación anidada impaciente:

	doble (doble 3)	
=	doble (3 + 3)	[def. de doble]
=	doble 6	[def. de +]
=	6 + 6	[def. de doble]
=	12	[def. de +]

- Ejemplo de evaluación anidada perezosa:

	doble (doble 3)	
=	(doble 3) + (doble 3)	[def. de doble]
=	(3 +3) + (doble 3)	[def. de doble]
=	6 + (doble 3)	[def. de +]
=	6 + (3 + 3)	[def. de doble]
=	6 + 6	[def. de +]
=	12	[def. de +]

Comprobación de propiedades

- Propiedad: El doble de x más y es el doble de x más el doble de y
- Expresión de la propiedad:

```
prop_doble x y = doble (x+y) == (doble x) + (doble y)
```

- Comprobación de la propiedad con QuickCheck:

```
*Main> quickCheck prop_doble
+++ OK, passed 100 tests.
```

- Para usar QuickCheck hay que importarlo, escribiendo al principio del fichero

```
import Test.QuickCheck
```

Refutación de propiedades

- Propiedad: El producto de dos números cualequiera es distinto de su suma.
- Expresión de la propiedad:

```
prop_prod_suma x y = x*y /= x+y
```

- Refutación de la propiedad con QuickCheck:

```
*Main> quickCheck prop_prod_suma
*** Failed! Falsifiable (after 1 test):
0
0
```

- Refinamiento: El producto de dos números no nulos cualequiera es distinto de su suma.

```
prop_prod_suma' x y =
    x /= 0 && y /= 0 ==> x*y /= x+y
```

- Refutación de la propiedad con QuickCheck:

```
*Main> quickCheck prop_prod_suma'
+++ OK, passed 100 tests.
*Main> quickCheck prop_prod_suma'
*** Failed! Falsifiable (after 5 tests):
2
2
```

1.2. Programación funcional

Programación funcional y programación imperativa

- La **programación funcional** es un estilo de programación cuyo método básico de computación es la aplicación de funciones a sus argumentos.
- Un **lenguaje de programación funcional** es uno que soporta y potencia el estilo funcional.
- La **programación imperativa** es un estilo de programación en el que los programas están formados por instrucciones que especifican cómo se ha de calcular el resultado.
- Ejemplo de problema para diferenciar los estilos de programación:
  Sumar los *n* primeros números.

Solución mediante programación imperativa

- Programa *suma n*:

  contador := 0
  total := 0
  repetir
    contador := contador + 1
    total := total + contador
  hasta que contador = n

- Evaluación de *suma 4*:

contador	total
0	0
1	1
2	3
3	6
4	10

Solución mediante programación funcional

- Programa:

```
suma n = sum [1..n]
```

- Evaluación de *suma 4*:

	suma 4	
=	sum [1..4]	[def. de suma]
=	sum [1, 2, 3, 4]	[def. de [..]]
=	1 + 2 + 3 + 4	[def. de sum]
=	10	[def. de +]

1.3. Rasgos característicos de Haskell

- Programas concisos.
- Sistema potente de tipos.
- Listas por comprensión.
- Funciones recursivas.
- Funciones de orden superior.
- Razonamiento sobre programas.
- Evaluación perezosa.
- Efectos monádicos.

1.4. Antecedentes históricos

- 1930s: Alonzo Church desarrolla el lambda cálculo (teoría básica de los lenguajes funcionales).
- 1950s: John McCarthy desarrolla el Lisp (lenguaje funcional con asignaciones).
- 1960s: Peter Landin desarrolla ISWIN (lenguaje funcional puro).
- 1970s: John Backus desarrolla FP (lenguaje funcional con orden superior).
- 1970s: Robin Milner desarrolla ML (lenguaje funcional con tipos polimórficos e inferencia de tipos).
- 1980s: David Turner desarrolla Miranda (lenguaje funcional perezoso).
- 1987: Un comité comienza el desarrollo de Haskell.
- 2003: El comité publica el "Haskell Report".

1.5. Presentación de Haskell

Ejemplo de recursión sobre listas

- Especificación: `(sum xs)` es la suma de los elementos de `xs`.
- Ejemplo: sum [2,3,7] $\leadsto$12
- Definición:

```
sum []     = 0
sum (x:xs) = x + sum xs
```

- Evaluación:

	sum [2,3,7]	
=	2 + sum [3,7]	[def. de sum]
=	2 + (3 + sum [7])	[def. de sum]
=	2 + (3 + (7 + sum []))	[def. de sum]
=	2 + (3 + (7 + 0))	[def. de sum]
=	12	[def. de +]

- Tipo de sum: `(Num a) => [a] -> a`

Ejemplo con listas de comprensión

- Especificación: `(ordena xs)` es la lista obtenida ordenando `xs` mediante el algoritmo de ordenación rápida.
- Ejemplo:

```
ordena [4,6,2,5,3] ⇝ [2,3,4,5,6]
ordena "deacb"     ⇝ "abcde"
```

- Definición:

```
ordena [] = []
ordena (x:xs) =
    (ordena menores) ++ [x] ++ (ordena mayores)
    where menores = [a | a <- xs, a <= x]
          mayores = [b | b <- xs, b > x]
```

- Tipo de `ordena`: `Ord a => [a] -> [a]`

Evaluación del ejemplo con listas de comprensión

	ordena [4,6,2,3]	
=	(ordena [2,3]) ++ [4] ++ (ordena [6])	[def. ordena]
=	((ordena []) ++ [2] ++ (ordena [3])) ++ [4] ++ (ordena [6])	[def. ordena]
=	([] ++ [2] ++ (ordena [3])) ++ [4] ++ (ordena [6])	[def. ordena]
=	([2] ++ (ordena [3])) ++ [4] ++ (ordena [6,5])	[def. ++]
=	([2] ++ ((ordena []) ++ [3] ++ [])) ++ [4] ++ (ordena [6])	[def. ordena]
=	([2] ++ ([] ++ [3] ++ [])) ++ [4] ++ (ordena [6])	[def. ordena]
=	([2] ++ [3]) ++ [4] ++ (ordena [6])	[def. ++]
=	[2,3] ++ [4] ++ (ordena [6])	[def. ++]
=	[2,3,4] ++ (ordena [6])	[def. ++]
=	[2,3,4] ++ ((ordena []) ++ [6] ++ (ordena []))	[def. ordena]
=	[2,3,4] ++ ((ordena []) ++ [6] ++ (ordena []))	[def. ordena]
=	[2,3,4] ++ ([] ++ [6] ++ [])	[def. ordena]
=	[2,3,4,6]	[def. ++]

Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
   - Cap. 1: Conceptos fundamentales.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.

- Cap. 1: Introduction.

3. B. O'Sullivan, D. Stewart y J. Goerzen *Real World Haskell*. O'Reilly, 2008.
   - Cap. 1: Getting Started.
4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
   - Cap. 1: Programación funcional.
5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
   - Cap. 1: Introducing functional programming.

Tema 2

Introducción a la programación con Haskell

2.1. El sistema GHC

El sistema GHC

- Los programa funcionales pueden evaluarse manualmente (como en el tema anterior).
- Los **lenguajes funcionales** evalúan automáticamente los programas funcionales.
- **Haskell** es un lenguaje funcional.
- **GHC** (Glasgow Haskell Compiler) es el intérprete de Haskell que usaremos en el curso.

2.2. Iniciación a GHC

2.2.1. Inicio de sesión con GHCi

- Inicio mediante `ghci`

```
I1M> ghci
GHCi, version 6.10.3: http://www.haskell.org/ghc/  :? for help
Prelude>
```

- La llamada es `Prelude>`
- Indica que ha cargado las definiciones básicas que forman el preludio y el sistema está listo para leer una expresión, evaluarla y escribir su resultado.

2.2.2. Cálculo aritmético

Cálculo aritmético: Operaciones aritméticas

- Operaciones aritméticas en Haskell:

```
Prelude> 2+3
5
Prelude> 2-3
-1
Prelude> 2*3
6
Prelude> 7 `div` 2
3
Prelude> 2^3
8
```

Cálculo aritmético: Precedencia y asociatividad

- Precedencia:

```
Prelude> 2*10^3
2000
Prelude> 2+3*4
14
```

- Asociacitividad:

```
Prelude> 2^3^4
2417851639229258349412352
Prelude> 2^(3^4)
2417851639229258349412352
Prelude> 2-3-4
-5
Prelude> (2-3)-4
-5
```

2.2.3. Cálculo con listas

Cálculo con listas: Seleccionar y eliminar

- Seleccionar el primer elemento de una lista no vacía:

```
head [1,2,3,4,5]  ⇝  1
```

- Eliminar el primer elemento de una lista no vacía:

```
tail [1,2,3,4,5]  ⇝  [2,3,4,5]
```

- Seleccionar el n–ésimo elemento de una lista (empezando en 0):

```
[1,2,3,4,5] !! 2  ⇝  3
```

- Seleccionar los n primeros elementos de una lista:

```
take 3 [1,2,3,4,5]  ⇝  [1,2,3]
```

- Eliminar los n primeros elementos de una lista:

```
drop 3 [1,2,3,4,5]  ⇝  [4,5]
```

Cálculo con listas

- Calcular la longitud de una lista:

```
length [1,2,3,4,5]  ⇝  5
```

- Calcular la suma de una lista de números:

```
sum [1,2,3,4,5]  ⇝  15
```

- Calcular el producto de una lista de números:

```
product [1,2,3,4,5]  ⇝  120
```

- Concatenar dos listas:

```
[1,2,3] ++ [4,5]  ⇝  [1,2,3,4,5]
```

- Invertir una lista:

```
reverse [1,2,3,4,5] ⇝  [5,4,3,2,1]
```

2.2.4. Cálculos con errores

Ejemplos de cálculos con errores

```
Prelude> 1 `div` 0
*** Exception: divide by zero
Prelude> head []
*** Exception: Prelude.head: empty list
Prelude> tail []
```

```
*** Exception: Prelude.tail: empty list
Prelude> [2,3] !! 5
*** Exception: Prelude.(!!): index too large
```

2.3. Aplicación de funciones

Aplicación de funciones en matemáticas y en Haskell

- Notación para funciones en matemáticas:
  - En matemáticas, la aplicación de funciones se representa usando paréntesis y la multiplicación usando yuxtaposición o espacios
  - Ejemplo:
    $f(a,b) + cd$
    representa la suma del valor de f aplicado a a y b más el producto de c por d.
- Notación para funciones en Haskell:
  - En Haskell, la aplicación de funciones se representa usando espacios y la multiplicación usando $*$.
  - Ejemplo:
    `f a b + c*d`
    representa la suma del valor de `f` aplicado a `a` y `b` más el producto de `c` por `d`.

Prioridad de la aplicación de funciones

- En Haskell, la aplicación de funciones tiene mayor prioridad que los restantes operadores. Por ejemplo, la expresión Haskell `f a + b` representa la expresión matemática $f(a) + b$.
- Ejemplos de expresiones Haskell y matemáticas:

Matemáticas	Haskell
$f(x)$	`f x`
$f(x,y)$	`f x y`
$f(g(x))$	`f (g x)`
$f(x,g(y))$	`f x (g y)`
$f(x)g(y)$	`f x * g y`

2.4. Guiones Haskell

- En Haskell los usuarios pueden definir funciones.
- Las nuevas definiciones se definen en guiones, que son ficheros de textos compuestos por una sucesión de definiciones.
- Se acostumbra a identificar los guiones de Haskell mediante el sufijo `.hs`

2.4.1. El primer guión Haskell

- Iniciar emacs y abrir dos ventanas: `C-x 2`
- En la primera ventana ejecutar Haskell: `M-x run-haskell`
- Cambiar a la otra ventana: `C-x o`
- Iniciar el guión: `C-x C-f ejemplo.hs`
- Escribir en el guión las siguientes definiciones

```
doble x     = x+x
cuadruple x = doble (doble x)
```

- Grabar el guión: `C-x C-s`
- Cargar el guión en Haskell: `C-c C-l`
- Evaluar ejemplos:

```
*Main> cuadruple 10
40
*Main> take (doble 2) [1,2,3,4,5,6]
[1,2,3,4]
```

- Volver al guión: `C-x o`
- Añadir al guión las siguientes definiciones:

```
factorial n = product [1..n]
media ns    = sum ns `div` length ns
```

- Grabar el guión: `C-x s`
- Cargar el guión en Haskell: `C-c C-l`

- Evaluar ejemplos:

```
*Main> factorial (doble 2)
24
*Main> doble (media [1,5,3])
6
```

2.4.2. Nombres de funciones

- Los nombres de funciones tienen que empezar por una letra en minúscula. Por ejemplo,
  - `sumaCuadrado`, `suma_cuadrado`, `suma'`
- Las palabras reservadas de Haskell no pueden usarse en los nombres de funciones. Algunas palabras reservadas son

```
case  class   data     default  deriving  do      else
if    import  in       infix    infixl    infixr  instance
let   module  newtype  of       then      type    where
```

- Se acostumbra escribir los argumentos que son listas usando `s` como sufijo de su nombre. Por ejemplo,
  - `ns` representa una lista de números,
  - `xs` representa una lista de elementos,
  - `css` representa una lista de listas de caracteres.

2.4.3. La regla del sangrado

- En Haskell la disposición del texto del programa (el **sangrado**) delimita las definiciones mediante la siguiente regla:

  > Una definición acaba con el primer trozo de código con un margen izquierdo menor o igual que el del comienzo de la definición actual.

- Ejemplo:

```
a = b + c
    where
      b = 1
      c = 2
d = a * 2
```

- Consejos:
  - Comenzar las definiciones de las funciones en la primera columna.
  - Usar el tabulador en emacs para determinar el sangrado en las definiciones.

2.4.4. Comentarios en Haskell

- En los guiones Haskell pueden incluirse comentarios.
- Un **comentario simple** comienza con `--` y se extiende hasta el final de la línea.
- Ejemplo de comentario simple:

```
-- (factorial n) es el factorial del número n.
factorial n = product [1..n]
```

- Un **comentario anidado** comienza con `{-` y termina en `-}`
- Ejemplo de comentario anidado:

```
{- (factorial n) es el factorial del número n.
   Por ejemplo, factorial 3  ==  6 -}
factorial n = product [1..n]
```

Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
   - Cap. 1: Conceptos fundamentales.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
   - Cap. 2: First steps.
3. B. O'Sullivan, D. Stewart y J. Goerzen *Real World Haskell*. O'Reilly, 2008.
   - Cap. 1: Getting Started.
4. B. Pope y A. van IJzendoorn *A Tour of the Haskell Prelude (basic functions)*
5. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
   - Cap. 2: Introducción a Haskell.

6. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.

   - Cap. 2: Getting started with Haskell and Hugs.

Tema 3

Tipos y clases

3.1. Conceptos básicos sobre tipos

¿Qué es un tipo?

- Un **tipo** es una colección de valores relacionados.
- Un ejemplo de tipos es el de los valores booleanos: `Bool`
- El tipo `Bool` tiene dos valores `True` (verdadero) y `False` (falso).
- `v :: T` representa que `v` es un valor del tipo `T` y se dice que "`v` tiene tipo `T`".
- Cálculo de tipo con `:type`

```
Prelude> :type True
True :: Bool
Prelude> :type False
False :: Bool
```

- El tipo `Bool -> Bool` está formado por todas las funciones cuyo argumento y valor son booleanos.
- Ejemplo de tipo `Bool -> Bool`

```
Prelude> :type not
not :: Bool -> Bool
```

Inferencia de tipos

- Regla de inferencia de tipos

$$\frac{f :: A \to B \quad e :: A}{f\ e :: B}$$

- Tipos de expresiones:

```
Prelude> :type not True

not True :: Bool
Prelude> :type not False

not False :: Bool
Prelude> :type not (not False)

not (not False) :: Bool
```

- Error de tipo:

```
Prelude> :type not 3
Error: No instance for (Num Bool)
Prelude> :type 1 + False
Error: No instance for (Num Bool)
```

Ventajas de los tipos

- Los lenguajes en los que la inferencia de tipo precede a la evaluación se denominan de **tipos seguros**.
- Haskell es un lenguaje de tipos seguros.
- En los lenguajes de tipos seguros no ocurren errores de tipos durante la evaluación.
- La inferencia de tipos no elimina todos los errores durante la evaluación. Por ejemplo,

```
Prelude> :type 1 `div` 0
1 `div` 0 :: (Integral t) => t
Prelude> 1 `div` 0
*** Exception: divide by zero
```

3.2. Tipos básicos

- `Bool` (**Valores lógicos**):
  - Sus valores son `True` y `False`.
- `Char` (**Caracteres**):

  - Ejemplos: 'a', 'B', '3', '+'
- String (**Cadena de caracteres**):
  - Ejemplos: "abc", "1 + 2 = 3"
- Int (**Enteros de precisión fija**):
  - Enteros entre -2^{31} y $2^{31} - 1$.
  - Ejemplos: 123, -12
- Integer (**Enteros de precisión arbitraria**):
  - Ejemplos: 1267650600228229401496703205376.
- Float (**Reales de precisión arbitraria**):
  - Ejemplos: 1.2, -23.45, 45e-7
- Double (**Reales de precisión doble**):
  - Ejemplos: 1.2, -23.45, 45e-7

3.3. Tipos compuestos

3.3.1. Tipos listas

- Una **lista** es una sucesión de elementos del mismo tipo.
- [T] es el tipo de las listas de elementos de tipo T.
- Ejemplos de listas:

```
[False, True]  :: [Bool]
['a','b','d']  :: [Char]
["uno","tres"] :: [String]
```

- Longitudes:
  - La **longitud** de una lista es el número de elementos.
  - La lista de longitud 0, [], es la **lista vacía**.
  - Las listas de longitud 1 se llaman **listas unitarias**.
- Comentarios:

- El tipo de una lista no informa sobre su longitud:

```
['a','b'] :: [Char]
['a','b','c'] :: [Char]
```

- El tipo de los elementos de una lista puede ser cualquiera:

```
[['a','b'],['c']] :: [[Char]]
```

3.3.2. Tipos tuplas

- Una **tupla** es una sucesión de elementos.
- $(T_1, T_2, \ldots, T_n)$ es el tipo de las n–tuplas cuya componente i–ésima es de tipo T_i.
- Ejemplos de tuplas:

```
(False,True)      :: (Bool,Bool)
(False,'a',True) :: (Bool,Char,Bool)
```

- Aridades:
  - La **aridad** de una tupla es el número de componentes.
  - La tupla de aridad 0, `()`, es la **tupla vacía**.
  - No están permitidas las tuplas de longitud 1.
- Comentarios:
  - El tipo de una tupla informa sobre su longitud:

```
('a','b')     :: (Char,Char)
('a','b','c') :: (Char,Char,Char)
```

  - El tipo de los elementos de una tupla puede ser cualquiera:

```
(('a','b'),['c','d']) :: ((Char,Char),[Char])
```

3.3.3. Tipos funciones

Tipos funciones

- Una **función** es una aplicación de valores de un tipo en valores de otro tipo.
- $T_1 \rightarrow T_2$ es el tipo de las funciones que aplica valores del tipo T_1 en valores del tipo T_2.
- Ejemplos de funciones:

```
not     :: Bool -> Bool
isDigit :: Char -> Bool
```

Funciones con múltiples argumentos o valores

- Ejemplo de función con múltiples argumentos:
  `suma (x,y)` es la suma de `x` e `y`. Por ejemplo, `suma (2,3)` es `5`.

```
suma :: (Int,Int) -> Int
suma (x,y) = x+y
```

- Ejemplo de función con múltiples valores:
  `deCeroA 5` es la lista de los números desde `0` hasta `n`. Por ejemplo, `deCeroA n` es `[0,1,2,3,4,5]`.

```
deCeroA :: Int -> [Int]
deCeroA n = [0..n]
```

- Notas:
  1. En las definiciones se ha escrito la **signatura** de las funciones.
  2. No es obligatorio escribir la signatura de las funciones.
  3. Es conveniente escribir las signatura.

3.4. Parcialización

Parcialización

- Mecanismo de **parcialización** (*currying* en inglés): Las funciones de más de un argumento pueden interpretarse como funciones que toman un argumento y devuelven otra función con un argumento menos.
- Ejemplo de parcialización:

```
suma' :: Int -> (Int -> Int)
suma' x y = x+y
```

`suma'` toma un entero `x` y devuelve la función `suma' x` que toma un entero `y` y devuelve la suma de `x` e `y`. Por ejemplo,

```
*Main> :type suma' 2
suma' 2 :: Int -> Int
*Main> :type suma' 2 3
suma' 2 3 :: Int
```

- Ejemplo de parcialización con tres argumentos:

```
mult :: Int -> (Int -> (Int -> Int))
mult x y z = x*y*z
```

`mult` toma un entero `x` y devuelve la función `mult x` que toma un entero `y` y devuelve la función `mult x y` que toma un entero `z` y devuelve `x*y*z`. Por ejemplo,

```
*Main> :type mult 2
mult 2 :: Int -> (Int -> Int)
*Main> :type mult 2 3
mult 2 3 :: Int -> Int
*Main> :type mult 2 3 7
mult 2 3 7 :: Int
```

Aplicación parcial

- Las funciones que toman sus argumentos de uno en uno se llaman **currificadas** (*curried* en inglés).
- Las funciones `suma'` y `mult` son currificadas.
- Las funciones currificadas pueden aplicarse parcialmente. Por ejemplo,

```
*Main> (suma' 2) 3
5
```

- Pueden definirse funciones usando aplicaciones parciales. Por ejemplo,

```
suc :: Int -> Int
suc = suma' 1
```

`suc x` es el sucesor de `x`. Por ejemplo, `suc 2` es `3`.

Convenios para reducir paréntesis

- Convenio 1: Las flechas en los tipos se asocia por la derecha. Por ejemplo,
  `Int -> Int -> Int -> Int`
  representa a
  `Int -> (Int -> (Int -> Int))`

- Convenio 2: Las aplicaciones de funciones se asocia por la izquierda. Por ejemplo,
  `mult x y z`
  representa a
  `((mult x) y) z`

- **Nota**: Todas las funciones con múltiples argumentos se definen en forma currificada, salvo que explícitamente se diga que los argumentos tienen que ser tuplas.

3.5. Polimorfismo y sobrecarga

3.5.1. Tipos polimórficos

- Un tipo es **polimórfico** ("tiene muchas formas") si contiene una variable de tipo.

- Una función es **polimórfica** si su tipo es polimórfico.

- La función `length` es polimófica:

  - Comprobación:

    ```
    Prelude> :type length
    length :: [a] -> Int
    ```

  - Significa que que para cualquier tipo `a`, `length` toma una lista de elementos de tipo `a` y devuelve un entero.
  - `a` es una variable de tipos.
  - Las variables de tipos tienen que empezar por minúscula.
  - Ejemplos:

    ```
    length [1, 4, 7, 1]                 ⇝  4
    length ["Lunes", "Martes", "Jueves"] ⇝  3
    length [reverse, tail]              ⇝  2
    ```

Ejemplos de funciones polimórficas

- `fst :: (a, b) -> a`

  ```
  fst (1,'x')       ⇝  1
  fst (True,"Hoy")  ⇝  True
  ```

- `head :: [a] -> a`

  ```
  head [2,1,4]    ⇝  2
  head ['b','c']  ⇝  'b'
  ```

- `take :: Int -> [a] -> [a]`

```
take 3 [3,5,7,9,4]        ⇝  [3,5,7]
take 2 ['l','o','l','a']  ⇝  "lo"
take 2 "lola"             ⇝  "lo"
```

- `zip :: [a] -> [b] -> [(a, b)]`

```
zip [3,5] "lo"  ⇝  [(3,'l'),(5,'o')]
```

- `id :: a -> a`

```
id 3    ⇝  3
id 'x'  ⇝  'x'
```

3.5.2. Tipos sobrecargados

- Un tipo está **sobrecargado** si contiene una restricción de clases.
- Una función está **sobregargada** si su tipo está sobrecargado.
- La función `sum` está sobrecargada:
  - Comprobación:

    ```
    Prelude> :type sum
    sum :: (Num a) => [a] -> a
    ```

  - Significa que que para cualquier tipo numérico `a`, `sum` toma una lista de elementos de tipo `a` y devuelve un valor de tipo `a`.
  - `Num a` es una restricción de clases.
  - Las restricciones de clases son expresiones de la forma `C a`, donde `C` es el nombre de una clase y `a` es una variable de tipo.
  - Ejemplos:

    ```
    sum [2, 3, 5]           ⇝  10
    sum [2.1, 3.23, 5.345]  ⇝  10.675
    ```

Ejemplos de tipos sobrecargados

- Ejemplos de funciones sobrecargadas:
  - `(-)    :: (Num a) => a -> a -> a`
  - `(*)    :: (Num a) => a -> a -> a`

- `negate :: (Num a) => a -> a`
- `abs    :: (Num a) => a -> a`
- `signum :: (Num a) => a -> a`

- Ejemplos de números sobrecargados:
  - `5   :: (Num t) => t`
  - `5.2 :: (Fractional t) => t`

3.6. Clases básicas

- Una **clase** es una colección de tipos junto con ciertas operaciones sobrecargadas llamadas **métodos**.

- Clases básicas:

`Eq`	tipos comparables por igualdad
`Ord`	tipos ordenados
`Show`	tipos mostrables
`Read`	tipos legibles
`Num`	tipos numéricos
`Integral`	tipos enteros
`Fractional`	tipos fraccionarios

La clase `Eq` (tipos comparables por igualdad)

- `Eq` contiene los tipos cuyos valores con comparables por igualdad.

- Métodos:

```
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

- Instancias:
  - `Bool`, `Char`, `String`, `Int`, `Integer`, `Float` y `Double`.
  - tipos compuestos: listas y tuplas.

- Ejemplos:

```
False == True       ⇝  False
False /= True       ⇝  True
'a' == 'b'          ⇝  False
"aei" == "aei"      ⇝  True
[2,3] == [2,3,2]    ⇝  False
('a',5) == ('a',5)  ⇝  True
```

La clase `Ord` (tipos ordenados)

- `Ord` es la subclase de `Eq` de tipos cuyos valores están ordenados.
- Métodos:

```
(<), (<=), (>), (>=) :: a -> a -> Bool
min, max             :: a -> a -> a
```

- Instancias:
  - `Bool`, `Char`, `String`, `Int`, `Integer`, `Float` y `Double`.
  - tipos compuestos: listas y tuplas.
- Ejemplos:

```
False < True              ⇝  True
min 'a' 'b'               ⇝  'a'
"elegante" < "elefante"   ⇝  False
[1,2,3] < [1,2]           ⇝  False
('a',2) < ('a',1)         ⇝  False
('a',2) < ('b',1)         ⇝  True
```

La clase `Show` (tipos mostrables)

- `Show` contiene los tipos cuyos valores se pueden convertir en cadenas de caracteres.
- Método:

```
show :: a -> String
```

- Instancias:
  - `Bool`, `Char`, `String`, `Int`, `Integer`, `Float` y `Double`.
  - tipos compuestos: listas y tuplas.
- Ejemplos:

```
show False       ~> "False"
show 'a'         ~> "'a'"
show 123         ~> "123"
show [1,2,3]     ~> "[1,2,3]"
show ('a',True)  ~> "('a',True)"
```

La clase `Read` (tipos legibles)

- `Read` contiene los tipos cuyos valores se pueden obtener a partir de cadenas de caracteres.
- Método:

```
read :: String -> a
```

- Instancias:
  - `Bool`, `Char`, `String`, `Int`, `Integer`, `Float` y `Double`.
  - tipos compuestos: listas y tuplas.
- Ejemplos:

```
read "False" :: Bool               ~> False
read "'a'" :: Char                 ~> 'a'
read "123" :: Int                  ~> 123
read "[1,2,3]" :: [Int]            ~> [1,2,3]
read "('a',True)" :: (Char,Bool)   ~> ('a',True)
```

La clase `Num` (tipos numéricos)

- `Num` es la subclase de `Eq` y `Ord` de tipos cuyos valores son números
- Métodos:

```
(+), (*), (-)         :: a -> a -> a
negate, abs, signum :: a -> a
```

- Instancias: `Int`, `Integer`, `Float` y `Double`.
- Ejemplos:

```
2+3           ~> 5
2.3+4.2       ~> 6.5
negate 2.7    ~> -2.7
abs (-5)      ~> 5
signum (-5)   ~> -1
```

La clase `Integral` (tipos enteros)

- `Integral` es la subclase de `Num` cuyo tipos tienen valores enteros.
- Métodos:

```
div :: a -> a -> a
mod :: a -> a -> a
```

- Instancias: `Int` e `Integer`.
- Ejemplos:

```
11 `div` 4  ⇝  2
11 `mod` 4  ⇝  3
```

La clase `Fractional` (tipos fraccionarios)

- `Fractional` es la subclase de `Num` cuyo tipos tienen valores no son enteros.
- Métodos:

```
(/)   :: a -> a -> a
recip :: a -> a
```

- Instancias: `Float` y `Double`.
- Ejemplos:

```
7.0 / 2.0  ⇝  3.5
recip 0.2  ⇝  5.0
```

Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
   - Cap. 2: Tipos de datos simples.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
   - Cap. 3: Types and classes.
3. B. O'Sullivan, D. Stewart y J. Goerzen *Real World Haskell*. O'Reilly, 2008.
   - Cap. 2: Types and Functions.

4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.

   - Cap. 2: Introducción a Haskell.
   - Cap. 5: El sistema de clases de Haskell.

5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.

   - Cap. 3: Basic types and definitions.

Tema 4

Definición de funciones

4.1. Definiciones por composición

- Decidir si un carácter es un dígito:

Prelude
```
isDigit :: Char -> Bool
isDigit c = c >= '0' && c <= '9'
```

- Decidir si un entero es par:

Prelude
```
even :: (Integral a) => a -> Bool
even n =  n `rem` 2 == 0
```

- Dividir una lista en su n–ésimo elemento:

Prelude
```
splitAt :: Int -> [a] -> ([a],[a])
splitAt n xs = (take n xs, drop n xs)
```

4.2. Definiciones con condicionales

- Calcular el valor absoluto (con condicionales):

Prelude
```
abs :: Int -> Int
abs n = if n >= 0 then n else -n
```

- Calcular el signo de un número (con condicionales anidados):

```
Prelude
signum :: Int -> Int
signum n = if n < 0 then (-1) else
              if n == 0 then 0 else 1
```

4.3. Definiciones con ecuaciones con guardas

- Calcular el valor absoluto (con ecuaciones guardadas):

```
Prelude
abs n | n >= 0    = n
      | otherwise = -n
```

- Calcular el signo de un número (con ecuaciones guardadas):

```
Prelude
signum n | n < 0     = -1
         | n == 0    = 0
         | otherwise = 1
```

4.4. Definiciones con equiparación de patrones

4.4.1. Constantes como patrones

- Calcular la negación:

```
Prelude
not :: Bool -> Bool
not True  =  False
not False =  True
```

- Calcular la conjunción (con valores):

```
Prelude
(&&)  :: Bool -> Bool -> Bool
True  && True  = True
True  && False = False
False && True  = False
False && False = False
```

4.4.2. Variables como patrones

- Calcular la conjunción (con variables anónimas):

Prelude
```
(&&) :: Bool -> Bool -> Bool
True  && True =  True
_     && _    =  False
```

- Calcular la conjunción (con variables):

Prelude
```
(&&) :: Bool -> Bool -> Bool
True  && x =  x
False && _ =  False
```

4.4.3. Tuplas como patrones

- Calcular el primer elemento de un par:

Prelude
```
fst :: (a,b) -> a
fst (x,_) = x
```

- Calcular el segundo elemento de un par:

Prelude
```
snd :: (a,b) -> b
snd (_,y) = y
```

4.4.4. Listas como patrones

- `(test1 xs)` se verifica si `xs` es una lista de 3 caracteres que empieza por `'a'`.

```
test1 :: [Char ] -> Bool
test1 ['a',_,_] = True
test1 _         = False
```

- Construcción de listas con `(:)`

```
[1,2,3] = 1:[2,3] = 1:(2:[3]) = 1:(2:(3:[]))
```

- `(test2 xs)` se verifica si `xs` es una lista de caracteres que empieza por `'a'`.

```
test2 :: [Char ] -> Bool
test2 ('a':_) = True
test2 _       = False
```

- Decidir si una lista es vacía:

Prelude
```
null :: [a] -> Bool
null []    = True
null (_:_) = False
```

- Primer elemento de una lista:

Prelude
```
head :: [a] -> a
head (x:_) =  x
```

- Resto de una lista:

Prelude
```
tail :: [a] -> [a]
tail (_:xs) = xs
```

4.4.5. Patrones enteros

- Predecesor de un número entero:

Prelude
```
pred :: Int -> Int
pred 0     = 0
pred (n+1) = n
```

- Comentarios sobre los patrones `n+k`:
  - `n+k` sólo se equipara con números mayores o iguales que `k`
  - Hay que escribirlo entre paréntesis.

4.5. Expresiones lambda

- Las funciones pueden construirse sin nombrarlas mediante las expresiones lambda.
- Ejemplo de evaluación de expresiones lambda:

```
Prelude> (\x -> x+x) 3
6
```

Uso de las expresiones lambda para resaltar la parcialización:

- (suma x y) es la suma de x e y.
- Definición sin lambda:

```
suma x y = x+y
```

- Definición con lambda:

```
suma' = \x -> (\y -> x+y)
```

Uso de las expresiones lambda en funciones como resultados:

- (const x y) es x.
- Definición sin lambda:

Prelude

```
const :: a -> b -> a
const x = x
```

- Definición con lambda:

```
const' :: a -> (b -> a)
const' x = \_ -> x
```

Uso de las expresiones lambda en funciones con sólo un uso:

- (impares n) es la lista de los n primeros números impares.
- Definición sin lambda:

```
impares n = map f [0..n-1]
    where f x = 2*x+1
```

- Definición con lambda:

```
impares' n = map (\x -> 2*x+1) [0..n-1]
```

4.6. Secciones

- Los **operadores** son las funciones que se escriben entre sus argumentos.
- Los operadores pueden convertirse en funciones prefijas escribiéndolos entre paréntesis.
- Ejemplo de conversión:

```
Prelude> 2 + 3
5
Prelude> (+) 2 3
5
```

- Ejemplos de secciones:

```
Prelude> (2+) 3
5
Prelude> (+3) 2
5
```

Expresión de secciones mediante lambdas

Sea $*$ un operador. Entonces

- `(*)  = \x -> (\y -> x*y)`
- `(x*) = \y -> x*y`
- `(*y) = \x -> x*y`

Aplicaciones de secciones

- Uso en definiciones de funciones mediante secciones

```
suma'     = (+)
siguiente = (1+)
inverso   = (1/)
doble     = (2*)
mitad     = (/2)
```

- Uso en signatura de operadores:

Prelude

```
(&&) :: Bool -> Bool -> Bool
```

- Uso como argumento:

```
Prelude> map (2*) [1..5]
[2,4,6,8,10]
```

Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
   - Cap. 1: Conceptos fundamentales.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
   - Cap. 4: Defining functions.
3. B. O'Sullivan, D. Stewart y J. Goerzen *Real World Haskell*. O'Reilly, 2008.
   - Cap. 2: Types and Functions.
4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
   - Cap. 2: Introducción a Haskell.
5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
   - Cap. 3: Basic types and definitions.

Tema 5

Definiciones de listas por comprensión

5.1. Generadores

Definiciones por comprensión

- Definiciones por comprensión en Matemáticas:
  $\{x^2 : x \in \{2,3,4,5\}\} = \{4,9,16,25\}$
- Definiciones por comprensión en Haskell:

```
Prelude> [x^2 | x <- [2..5]]
[4,9,16,25]
```

- La expresión `x <- [2..5]` se llama un **generador**.
- Ejemplos con más de un generador:

```
Prelude> [(x,y) | x <- [1,2,3], y <- [4,5]]
[(1,4),(1,5),(2,4),(2,5),(3,4),(3,5)]
Prelude> [(x,y) | y <- [4,5], x <- [1,2,3]]
[(1,4),(2,4),(3,4),(1,5),(2,5),(3,5)]
```

Generadores dependientes

- Ejemplo con generadores dependientes:

```
Prelude> [(x,y) | x <- [1..3], y <- [x..3]]
[(1,1),(1,2),(1,3),(2,2),(2,3),(3,3)]
```

- `(concat xss)` es la concatenación de la lista de listas `xss`. Por ejemplo,

```
concat [[1,3],[2,5,6],[4,7]]  ⇝  [1,3,2,5,6,4,7]
```

Prelude
```
concat :: [[a]] -> [a]
concat xss = [x | xs <- xss, x <- xs]
```

Generadores con variables anónimas

- Ejemplo de generador con variable anónima:
  (primeros ps) es la lista de los primeros elementos de la lista de pares ps. Por ejemplo,

  ```
  primeros [(1,3),(2,5),(6,3)]  ⇝  [1,2,6]
  ```

  ```
  primeros :: [(a, b)] -> [a]
  primeros ps =  [x | (x,_) <- ps]
  ```

- Definición de la longitud por comprensión

  Prelude
  ```
  length :: [a] -> Int
  length xs = sum [1 | _ <- xs]
  ```

5.2. Guardas

- Las listas por comprensión pueden tener **guardas** para restringir los valores.
- Ejemplo de guarda:

  ```
  Prelude> [x | x <- [1..10], even x]
  [2,4,6,8,10]
  ```

  La guarda es even x.

- (factores n) es la lista de los factores del número n. Por ejemplo,

  ```
  factores 30  ⇝  [1,2,3,5,6,10,15,30]
  ```

  ```
  factores :: Int -> [Int]
  factores n = [x | x <- [1..n], n `mod` x == 0]
  ```

- (primo n) se verifica si n es primo. Por ejemplo,

```
primo 30  ↝ False
primo 31  ↝ True
```

```
primo :: Int -> Bool
primo n = factores n == [1, n]
```

- (`primos n`) es la lista de los primos menores o iguales que `n`. Por ejemplo,

```
primos 31  ↝ [2,3,5,7,11,13,17,19,23,29,31]
```

```
primos :: Int -> [Int]
primos n = [x | x <- [2..n], primo x]
```

Guarda con igualdad

- Una **lista de asociación** es una lista de pares formado por una clave y un valor. Por ejemplo,

```
[("Juan",7),("Ana",9),("Eva",3)]
```

- (`busca c t`) es la lista de los valores de la lista de asociación `t` cuyas claves valen `c`. Por ejemplo,

```
Prelude> busca 'b' [('a',1),('b',3),('c',5),('b',2)]
[3,2]
```

```
busca :: Eq a => a -> [(a, b)] -> [b]
busca c t = [v | (c', v) <- t, c' == c]
```

5.3. La función `zip`

La función `zip` y elementos adyacentes

- (`zip xs ys`) es la lista obtenida emparejando los elementos de las listas `xs` e `ys`. Por ejemplo,

```
Prelude> zip ['a','b','c'] [2,5,4,7]
[('a',2),('b',5),('c',4)]
```

- (`adyacentes xs`) es la lista de los pares de elementos adyacentes de la lista xs. Por ejemplo,

```
adyacentes [2,5,3,7]  ⇝  [(2,5),(5,3),(3,7)]
```

```
adyacentes :: [a] -> [(a, a)]
adyacentes xs = zip xs (tail xs)
```

Las funciones `zip`, `and` y listas ordenadas

- (`and xs`) se verifica si todos los elementos de `xs` son verdaderos. Por ejemplo,

```
and [2 < 3, 2+3 == 5]         ⇝  True
and [2 < 3, 2+3 == 5, 7 < 7]  ⇝  False
```

- (`ordenada xs`) se verifica si la lista `xs` está ordenada. Por ejemplo,

```
ordenada [1,3,5,6,7]  ⇝  True
ordenada [1,3,6,5,7]  ⇝  False
```

```
ordenada :: Ord a => [a] -> Bool
ordenada xs = and [x <= y | (x,y) <- adyacentes xs]
```

La función `zip` y lista de posiciones

- (`posiciones x xs`) es la lista de las posiciones ocupadas por el elemento `x` en la lista `xs`. Por ejemplo,

```
posiciones 5 [1,5,3,5,5,7]  ⇝  [1,3,4]
```

```
posiciones :: Eq a => a -> [a] -> [Int]
posiciones x xs =
    [i | (x',i) <- zip xs [0..n], x == x']
    where n = length xs - 1
```

5.4. Comprensión de cadenas

Cadenas y listas

- Las cadenas son listas de caracteres. Por ejemplo,

```
*Main> "abc" == ['a','b','c']
True
```

- La expresión

```
"abc" :: String
```

es equivalente a

```
['a','b','c'] :: [Char]
```

- Las funciones sobre listas se aplican a las cadenas:

```
length "abcde"               ⇝  5
reverse "abcde"              ⇝  "edcba"
"abcde" ++ "fg"              ⇝  "abcdefg"
posiciones 'a' "Salamanca"   ⇝  [1,3,5,8]
```

Definiciones sobre cadenas con comprensión

- (minusculas c) es la cadena formada por las letras minúsculas de la cadena c. Por ejemplo,

```
minusculas "EstoEsUnaPrueba"  ⇝  "stosnarueba"
```

```
minusculas :: String -> String
minusculas xs = [x | x <- xs, elem x ['a'..'z']]
```

- (ocurrencias x xs) es el número de veces que ocurre el carácter x en la cadena xs. Por ejemplo,

```
ocurrencias 'a' "Salamanca"  ⇝  4
```

```
ocurrencias :: Char -> String -> Int
ocurrencias x xs = length [x' | x' <- xs, x == x']
```

5.5. Cifrado César

- En el cifrado César cada letra en el texto original es reemplazada por otra letra que se encuentra 3 posiciones más adelante en el alfabeto.
- La codificación de

```
"en todo la medida"
```

es

```
"hq wrgr od phglgd"
```

- Se puede generalizar desplazando cada letra n posiciones.
- La codificación con un desplazamiento 5 de

  ```
  "en todo la medida"
  ```

  es

  ```
  "js ytit qf rjinif"
  ```

- La descodificación de un texto codificado con un desplazamiento n se obtiene codificándolo con un desplazamiento $-n$.

5.5.1. Codificación y descodificación

Las funciones `ord` y `char`

- `(ord c)` es el código del carácter `c`. Por ejemplo,

  ```
  ord 'a'  ⇝  97
  ord 'b'  ⇝  98
  ord 'A'  ⇝  65
  ```

- `(char n)` es el carácter de código `n`. Por ejemplo,

  ```
  chr 97  ⇝  'a'
  chr 98  ⇝  'b'
  chr 65  ⇝  'A'
  ```

Codificación y descodificación: Código de letra

- Simplificación: Sólo se codificarán las letras minúsculas dejando los restantes caracteres sin modificar.
- `(let2int c)` es el entero correspondiente a la letra minúscula `c`. Por ejemplo,

  ```
  let2int 'a'  ⇝  0
  let2int 'd'  ⇝  3
  let2int 'z'  ⇝  25
  ```

  ```
  let2int :: Char -> Int
  let2int c = ord c - ord 'a'
  ```

Codificación y descodificación: Letra de código

- (`int2let n`) es la letra minúscula correspondiente al entero `n`. Por ejemplo,

```
int2let 0  ~>  'a'
int2let 3  ~>  'd'
int2let 25 ~>  'z'
```

```
int2let :: Int -> Char
int2let n = chr (ord 'a' + n)
```

Codificación y descodificación: Desplazamiento

- (`desplaza n c`) es el carácter obtenido desplazando `n` caracteres el carácter `c`. Por ejemplo,

```
desplaza   3  'a'  ~>  'd'
desplaza   3  'y'  ~>  'b'
desplaza (-3) 'd'  ~>  'a'
desplaza (-3) 'b'  ~>  'y'
```

```
desplaza :: Int -> Char -> Char
desplaza n c
    | elem c ['a'..'z'] = int2let ((let2int c+n) `mod` 26)
    | otherwise         = c
```

Codificación y descodificación

- (`codifica n xs`) es el resultado de codificar el texto `xs` con un desplazamiento `n`. Por ejemplo,

```
Prelude> codifica   3  "En todo la medida"
"Eq wrgr od phglgd"
Prelude> codifica (-3) "Eq wrgr od phglgd"
"En todo la medida"
```

```
codifica :: Int -> String -> String
codifica n xs = [desplaza n x | x <- xs]
```

Propiedades de la codificación con QuickCheck

- Propiedad: Al desplazar $-n$ un carácter desplazado n, se obtiene el carácter inicial.

```
prop_desplaza n xs =
    desplaza (-n) (desplaza n xs) == xs
```

```
*Main> quickCheck prop_desplaza
+++ OK, passed 100 tests.
```

- Propiedad: Al codificar con $-n$ una cadena codificada con n, se obtiene la cadena inicial.

```
prop_codifica n xs =
    codifica (-n) (codifica n xs) == xs
```

```
*Main> quickCheck prop_codifica
+++ OK, passed 100 tests.
```

5.5.2. Análisis de frecuencias

Tabla de frecuencias

- Para descifrar mensajes se parte de la frecuencia de aparición de letras.
- `tabla` es la lista de la frecuencias de las letras en castellano, Por ejemplo, la frecuencia de la `'a'` es del 12.53 %, la de la `'b'` es 1.42 %.

```
tabla :: [Float]
tabla = [12.53, 1.42, 4.68, 5.86, 13.68, 0.69, 1.01,
          0.70, 6.25, 0.44, 0.01,  4.97, 3.15, 6.71,
          8.68, 2.51, 0.88, 6.87,  7.98, 4.63, 3.93,
          0.90, 0.02, 0.22, 0.90,  0.52]
```

Frecuencias

- `(porcentaje n m)` es el porcentaje de `n` sobre `m`. Por ejemplo,

```
porcentaje 2 5  ⇝  40.0
```

```
porcentaje :: Int -> Int -> Float
porcentaje n m = (fromIntegral n / fromIntegral m) * 100
```

- `(frecuencias xs)` es la frecuencia de cada una de las minúsculas de la cadena `xs`. Por ejemplo,

```
Prelude> frecuencias "en todo la medida"
[14.3,0,0,21.4,14.3,0,0,0,7.1,0,0,7.1,
 7.1,7.1,14.3,0,0,0,0,7.1,0,0,0,0,0,0]
```

```
frecuencias :: String -> [Float]
frecuencias xs =
    [porcentaje (ocurrencias x xs) n | x <- ['a'..'z']]
    where n = length (minusculas xs)
```

5.5.3. Descifrado

Descifrado: Ajuste chi cuadrado

- Una medida de la discrepancia entre la distribución observada os_i y la esperada es_i es

$$\chi^2 = \sum_{i=0}^{n-1} \frac{(os_i - es_i)^2}{es_i}$$

  Los menores valores corresponden a menores discrepancias.

- `(chiCuad os es)` es la medida chi cuadrado de las distribuciones `os` y `es`. Por ejemplo,

```
chiCuad [3,5,6] [3,5,6]  ⇝  0.0
chiCuad [3,5,6] [5,6,3]  ⇝  3.9666667
```

```
chiCuad :: [Float] -> [Float] -> Float
chiCuad os es =
    sum [((o-e)^2)/e | (o,e) <- zip os es]
```

Descifrado: Rotación

- `(rota n xs)` es la lista obtenida rotando `n` posiciones los elementos de la lista `xs`. Por ejemplo,

```
rota 2 "manolo"  ⇝  "noloma"
```

```
rota :: Int -> [a] -> [a]
rota n xs = drop n xs ++ take n xs
```

Descifrado

- (descifra xs) es la cadena obtenida descodificando la cadena xs por el anti-desplazamiento que produce una distribución de minúsculas con la menor desviación chi cuadrado respecto de la tabla de distribución de las letras en castellano. Por ejemplo,

```
*Main> codifica 5 "Todo para nada"
"Ttit ufwf sfif"
*Main> descifra "Ttit ufwf sfif"
"Todo para nada"
```

```
descifra :: String -> String
descifra xs =  codifica (-factor) xs
 where
  factor = head (posiciones (minimum tabChi) tabChi)
  tabChi = [chiCuad (rota n tabla') tabla | n <- [0..25]]
  tabla' = frecuencias xs
```

Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
   - Cap. 4: Listas.
2. G. Hutton *Programming in Haskell*. Cambridge University Press.
   - Cap. 5: List comprehensions.
3. B. O'Sullivan, D. Stewart y J. Goerzen *Real World Haskell*. O'Reilly, 2008.
   - Cap. 12: Barcode Recognition.
4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
   - Cap. 6: Programación con listas.
5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
   - Cap. 5: Data types: tuples and lists.

Tema 6

Funciones recursivas

6.1. Recursión numérica

Recursión numérica: El factorial

- La función factorial:

```
factorial :: Integer -> Integer
factorial 0       = 1
factorial (n + 1) = (n + 1) * factorial n
```

- Cálculo:

```
factorial 3 = 3 * (factorial 2)
            = 3 * (2 * (factorial 1))
            = 3 * (2 * (1 * (factorial 0)))
            = 3 * (2 * (1 * 1))
            = 3 * (2 * 1)
            = 3 * 2
            = 6
```

Recursión numérica: El producto

- Definición recursiva del producto:

```
por :: Int -> Int -> Int
m `por` 0       = 0
m `por` (n + 1) = m + (m `por` n)
```

- Cálculo:

```
3 ‘por‘ 2 = 3 + (3 ‘por‘ 1)
          = 3 + (3 + (3 ‘por‘ 0))
          = 3 + (3 + 0)
          = 3 + 3
          = 6
```

6.2. Recusión sobre lista

Recursión sobre listas: La función `product`

- Producto de una lista de números:

Prelude
```
product :: Num a => [a] -> a
product []     = 1
product (n:ns) = n * product ns
```

- Cálculo:

```
product [7,5,2] = 7 * (product [5,2])
                = 7 * (5 * (product [2]))
                = 7 * (5 * (2 * (product [])))
                = 7 * (5 * (2 * 1))
                = 7 * (5 * 2)
                = 7 * 10
                = 70
```

Recursión sobre listas: La función `length`

- Longitud de una lista:

Prelude
```
length :: [a] -> Int
length []     = 0
length (_:xs) = 1 + length xs
```

- Cálculo:

```
length [2,3,5] = 1 + (length [3,5])
               = 1 + (1 + (length [5]))
               = 1 + (1 + (1 + (length [])))
               = 1 + (1 + (1 + 0))
```

```
                  = 1 + (1 + 1)
                  = 1 + 2
                  = 3
```

Recursión sobre listas: La función `reverse`

- Inversa de una lista:

Prelude
```
reverse :: [a] -> [a]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

- Cálculo:

```
reverse [2,5,3] = (reverse [5,3]) ++ [2]
                = ((reverse [3]) ++ [5]) ++ [2]
                = (((reverse []) ++ [3]) ++ [5]) ++ [2]
                = (([] ++ [3]) ++ [5]) ++ [2]
                = ([3] ++ [5]) ++ [2]
                = [3,5] ++ [2]
                = [3,5,2]
```

Recursión sobre listas: `++`

- Concatenación de listas:

Prelude
```
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys
(x:xs) ++ ys = x : (xs ++ ys)
```

- Cálculo:

```
[1,3,5] ++ [2,4] = 1:([3,5] ++ [2,4])
                 = 1:(3:([5] ++ [2,4]))
                 = 1:(3:(5:([] ++ [2,4])))
                 = 1:(3:(5:[2,4]))
                 = 1:(3:[5,2,4])
                 = 1:[3,5,2,4]
                 = [1,3,5,2,4]
```

Recursión sobre listas: Inserción ordenada

- `(inserta e xs)` inserta el elemento `e` en la lista `xs` delante del primer elemento de `xs` mayor o igual que `e`. Por ejemplo,

```
inserta 5 [2,4,7,3,6,8,10] ⇝ [2,4,5,7,3,6,8,10]
```

```
inserta :: Ord a => a -> [a] -> [a]
inserta e []                  = [e]
inserta e (x:xs) | e <= x     = e : (x:xs)
                 | otherwise  = x : inserta e xs
```

- Cálculo:

```
inserta 4 [1,3,5,7] = 1:(inserta 4 [3,5,7])
                    = 1:(3:(inserta 4 [5,7]))
                    = 1:(3:(4:(5:[7])))
                    = 1:(3:(4:[5,7]))
                    = [1,3,4,5,7]
```

Recursión sobre listas: Ordenación por inserción

- `(ordena_por_insercion xs)` es la lista `xs` ordenada mediante inserción, Por ejemplo,

```
ordena_por_insercion [2,4,3,6,3] ⇝ [2,3,3,4,6]
```

```
ordena_por_insercion :: Ord a => [a] -> [a]
ordena_por_insercion []     = []
ordena_por_insercion (x:xs) =
    inserta x (ordena_por_insercion xs)
```

- Cálculo:

```
  ordena_por_insercion [7,9,6] =
= inserta 7 (inserta 9 (inserta 6 []))
= inserta 7 (inserta 9 [6])
= inserta 7 [6,9]
= [6,7,9]
```

6.3. Recursión sobre varios argumentos

Recursión sobre varios argumentos: La función `zip`

- Emparejamiento de elementos (la función `zip`):

Prelude
```
zip :: [a] -> [b] -> [(a, b)]
zip []     _      = []
zip _      []     = []
zip (x:xs) (y:ys) = (x,y) : zip xs ys
```

- Cálculo:

```
  zip [1,3,5] [2,4,6,8]
= (1,2) : (zip [3,5] [4,6,8])
= (1,2) : ((3,4) : (zip [5] [6,8]))
= (1,2) : ((3,4) : ((5,6) : (zip [] [8])))
= (1,2) : ((3,4) : ((5,6) : []))
= [(1,2),(3,4),(5,6)]
```

Recursión sobre varios argumentos: La función `drop`

- Eliminación de elementos iniciales:

Prelude
```
drop :: Int -> [a] -> [a]
drop 0 xs          = xs
drop (n+1) []      = []
drop (n+1) (x:xs) = drop n xs
```

- Cálculo:

```
  drop 2 [5,7,9,4]     |          drop 5 [1,4]
= drop 1 [7,9,4]       |        = drop 4 [4]
= drop 0 [9,4]         |        = drop 1 []
= [9,4]                |        = []
```

6.4. Recursión múltiple

Recursión múltiple: La función de Fibonacci

- La sucesión de Fibonacci es: 0,1,1,2,3,5,8,13,21,.... Sus dos primeros términos son 0 y 1 y los restantes se obtienen sumando los dos anteriores.

- `(fibonacci n)` es el `n`–ésimo término de la sucesión de Fibonacci. Por ejemplo,

```
fibonacci 8  ⇝  21
```

```
fibonacci :: Int -> Int
fibonacci 0     = 0
fibonacci 1     = 1
fibonacci (n+2) = fibonacci n + fibonacci (n+1)
```

Recursión múltiple: Ordenación rápida

- Algoritmo de ordenación rápida:

```
ordena :: (Ord a) => [a] -> [a]
ordena [] = []
ordena (x:xs) =
    (ordena menores) ++ [x] ++ (ordena mayores)
    where menores = [a | a <- xs, a <= x]
          mayores = [b | b <- xs, b > x]
```

6.5. Recursión mutua

Recursión mutua: Par e impar

- Par e impar por recursión mutua:

```
par :: Int -> Bool
par 0     = True
par (n+1) = impar n

impar :: Int -> Bool
impar 0      = False
impar (n+1)  = par n
```

- Cálculo:

```
  impar 3               |           par 3
= par 2                 |         = impar 2
= impar 1               |         = par 1
= par 0                 |         = impar 0
= True                  |         = False
```

Recursión mutua: Posiciones pares e impares

- `(pares xs)` son los elementos de `xs` que ocupan posiciones pares.
- `(impares xs)` son los elementos de `xs` que ocupan posiciones impares.

```
pares :: [a] -> [a]
pares []     = []
pares (x:xs) = x : impares xs

impares :: [a] -> [a]
impares []     = []
impares (_:xs) = pares xs
```

- Cálculo:

```
  pares [1,3,5,7]
= 1:(impares [3,5,7])
= 1:(pares [5,7])
= 1:(5:(impares [7]))
= 1:(5:[])
= [1,5]
```

6.6. Heurísticas para las definiciones recursivas

Aplicación del método: La función `product`

- Paso 1: Definir el tipo:

```
product :: [Int] -> Int
```

- Paso 2: Enumerar los casos:

```
product :: [Int] -> Int
product []     =
product (n:ns) =
```

- Paso 3: Definir los casos simples:

```
product :: [Int] -> Int
product []     = 1
product (n:ns) =
```

- Paso 4: Definir los otros casos:

```
product :: [Int] -> Int
product []     = 1
product (n:ns) = n * product ns
```

- Paso 5: Generalizar y simplificar:

```
product :: Num a => [a] -> a
product = foldr (*) 1
```

donde `(foldr op e l)` pliega por la derecha la lista `l` colocando el operador `op` entre sus elementos y el elemento `e` al final. Por ejemplo,

```
foldr (+) 6 [2,3,5] ⇝ 2+(3+(5+6)) ⇝ 16
foldr (-) 6 [2,3,5] ⇝ 2-(3-(5-6)) ⇝ -2
```

Aplicación del método: La función `drop`

- Paso 1: Definir el tipo:

```
drop :: Int -> [a] -> [a]
```

- Paso 2: Enumerar los casos:

```
drop :: Int -> [a] -> [a]
drop 0 []          =
drop 0 (x:xs)      =
drop (n+1) []      =
drop (n+1) (x:xs) =
```

- Paso 3: Definir los casos simples:

```
drop :: Int -> [a] -> [a]
drop 0 []          = []
drop 0 (x:xs)      = x:xs
drop (n+1) []      = []
drop (n+1) (x:xs) =
```

- Paso 4: Definir los otros casos:

```
drop :: Int -> [a] -> [a]
drop 0 []          = []
drop 0 (x:xs)      = x:xs
drop (n+1) []      = []
drop (n+1) (x:xs) = drop n xs
```

- Paso 5: Generalizar y simplificar:

```
drop :: Integral b => b -> [a] -> [a]
drop 0 xs          = xs
drop (n+1) []      = []
drop (n+1) (_:xs) = drop n xs
```

Aplicación del método: La función `init`

- `init` elimina el último elemento de una lista no vacía.
- Paso 1: Definir el tipo:

```
init :: [a] -> [a]
```

- Paso 2: Enumerar los casos:

```
init :: [a] -> [a]
init (x:xs) =
```

- Paso 3: Definir los casos simples:

```
init :: [a] -> [a]
init (x:xs) | null xs   = []
            | otherwise =
```

- Paso 4: Definir los otros casos:

```
init :: [a] -> [a]
init (x:xs) | null xs   = []
            | otherwise = x : init xs
```

- Paso 5: Generalizar y simplificar:

```
init :: [a] -> [a]
init [_]    = []
init (x:xs) = x : init xs
```

Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
   - Cap. 3: Números.
   - Cap. 4: Listas.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
   - Cap. 6: Recursive functions.
3. B. O'Sullivan, D. Stewart y J. Goerzen *Real World Haskell*. O'Reilly, 2008.
   - Cap. 2: Types and Functions.
4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
   - Cap. 2: Introducción a Haskell.
   - Cap. 6: Programación con listas.
5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
   - Cap. 4: Designing and writing programs.

Tema 7

Funciones de orden superior

7.1. Funciones de orden superior

Funciones de orden superior

- Una función es **de orden superior** si toma una función como argumento o devuelve una función como resultado.
- `(dosVeces f x)` es el resultado de aplicar `f` a `f x`. Por ejemplo,

```
dosVeces (*3) 2            ⇝  18
dosVeces reverse [2,5,7]   ⇝  [2,5,7]
```

```
dosVeces :: (a -> a) -> a -> a
dosVeces f x = f (f x)
```

- Prop: `dosVeces reverse = id`
  donde `id` es la función identidad.

Prelude

```
id :: a -> a
id x =  x
```

Usos de las funciones de orden superior

- Definición de patrones de programación.
  - Aplicación de una función a todos los elementos de una lista.
  - Filtrado de listas por propiedades.
  - Patrones de recursión sobre listas.

- Diseño de lenguajes de dominio específico:
  - Lenguajes para procesamiento de mensajes.
  - Analizadores sintácticos.
  - Procedimientos de entrada/salida.
- Uso de las propiedades algebraicas de las funciones de orden superior para razonar sobre programas.

7.2. Procesamiento de listas

7.2.1. La función `map`

La función `map`: Definición

- `(map f xs)` es la lista obtenida aplicando `f` a cada elemento de `xs`. Por ejemplo,

```
map (*2) [3,4,7]     ~> [6,8,14]
map sqrt [1,2,4]     ~> [1.0,1.4142135623731,2.0]
map even [1..5]      ~> [False,True,False,True,False]
```

- Definición de `map` por comprensión:

```haskell
map :: (a -> b) -> [a] -> [b]
map f xs =  [f x | x <- xs]
```

- Definición de `map` por recursión:

Prelude
```haskell
map :: (a -> b) -> [a] -> [b]
map _ []     = []
map f (x:xs) = f x : map f xs
```

Relación entre `sum` y `map`

- La función `sum`:

Prelude
```haskell
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

- Propiedad: `sum (map (2*) xs) = 2 * sum xs`

- Comprobación con QuickCheck:

```
prop_sum_map :: [Int] -> Bool
prop_sum_map xs = sum (map (2*) xs) == 2 * sum xs
```

```
*Main> quickCheck prop_sum_map
+++ OK, passed 100 tests.
```

7.2.2. La función `filter`

La función `filter`

- `filter p xs` es la lista de los elementos de `xs` que cumplen la propiedad `p`. Por ejemplo,

```
filter even [1,3,5,4,2,6,1] ⇝ [4,2,6]
filter (>3) [1,3,5,4,2,6,1] ⇝ [5,4,6]
```

- Definición de `filter` por comprensión:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p xs =  [x | x <- xs, p x]
```

- Definición de `filter` por recursión:

Prelude
```
filter :: (a -> Bool) -> [a] -> [a]
filter _ []                    = []
filter p (x:xs) | p x          = x : filter p xs
                | otherwise = filter p xs
```

Uso conjunto de `map` y `filter`

- `sumaCuadradosPares xs` es la suma de los cuadrados de los números pares de la lista `xs`. Por ejemplo,

```
sumaCuadradosPares [1..5]  ⇝  20
```

```
sumaCuadradosPares :: [Int] -> Int
sumaCuadradosPares xs = sum (map (^2) (filter even xs))
```

- Definición por comprensión:

```
sumaCuadradosPares' :: [Int] -> Int
sumaCuadradosPares' xs = sum [x^2 | x <- xs, even x]
```

Predefinidas de orden superior para procesar listas

- `all p xs` se verifica si todos los elementos de `xs` cumplen la propiedad `p`. Por ejemplo,

```
all odd [1,3,5]  ⇝  True
all odd [1,3,6]  ⇝  False
```

- `any p xs` se verifica si algún elemento de `xs` cumple la propiedad `p`. Por ejemplo,

```
any odd [1,3,5]  ⇝  True
any odd [2,4,6]  ⇝  False
```

- `takeWhile p xs` es la lista de los elementos iniciales de `xs` que verifican el predicado `p`. Por ejemplo,

```
takeWhile even [2,4,6,7,8,9] ⇝ [2,4,6]
```

- `dropWhile p xs` es la lista `xs` sin los elementos iniciales que verifican el predicado `p`. Por ejemplo,

```
dropWhile even [2,4,6,7,8,9] ⇝ [7,8,9]
```

7.3. Función de plegado por la derecha: `foldr`

Esquema básico de recursión sobre listas

- Ejemplos de definiciones recursivas:

Prelude

```
sum []          = 0
sum (x:xs)      = x + sum xs
product []      = 1
product (x:xs)  = x * product xs
or []           = False
or (x:xs)       = x || or xs
and []          = True
and (x:xs)      = x && and xs
```

- Esquema básico de recursión sobre listas:

```
f []     = v
f (x:xs) = x `op` (f xs)
```

El patrón `foldr`

- Redefiniciones con el patrón `foldr`

```
sum     = foldr (+) 0
product = foldr (*) 1
or      = foldr (||) False
and     = foldr (&&) True
```

- Definición del patrón `foldr`

Prelude

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f v []     =  v
foldr f v (x:xs) =  f x (foldr f v xs)
```

Visión no recursiva de `foldr`

- Cálculo con `sum`:

```
sum [2,3,5]
= foldr (+) 0 [2,3,5]          [def. de sum]
= foldr (+) 0 2:(3:(5:[]))     [notación de lista]
=             2+(3+(5+0))      [sustituir (:) por (+) y [] por 0]
= 10                           [aritmética]
```

- Cálculo con `sum`:

```
product [2,3,5]
= foldr (*) 1 [2,3,5]          [def. de sum]
= foldr (*) 1 2:(3:(5:[]))     [notación de lista]
=             2*(3*(5*1))      [sustituir (:) por (*) y [] por 1]
= 30                           [aritmética]
```

- Cálculo de `foldr f v xs`
  Sustituir en `xs` los `(:)` por `f` y `[]` por `v`.

Definición de la longitud mediante `foldr`

- Ejemplo de cálculo de la longitud:

```
  longitud [2,3,5]
= longitud 2:(3:(5:[]))
=          1+(1+(1+0))      [Sustituciones]
= 3
```

- Sustituciones:
  - los `(:)` por `(\_ n -> 1+n)`
  - la `[]` por `0`
- Definición de `length` usando `foldr`

```
longitud :: [a] -> Int
longitud = foldr (\_ n -> 1+n) 0
```

Definición de la inversa mediante `foldr`

- Ejemplo de cálculo de la inversa:

```
  inversa [2,3,5]
= inversa 2:(3:(5:[]))
=         (([] ++ [5]) ++ [3]) ++ [2]  [Sustituciones]
= [5,3,2]
```

- Sustituciones:
  - los `(:)` por `(\x xs -> xs ++ [x])`
  - la `[]` por `[]`
- Definición de `inversa` usando `foldr`

```
inversa :: [a] -> [a]
inversa = foldr (\x xs -> xs ++ [x]) []
```

Definición de la concatenación mediante `foldr`

- Ejemplo de cálculo de la concatenación:

```
  conc [2,3,5] [7,9]
= conc 2:(3:(5:[])) [7,9]
=      2:(3:(5:[7,9]))       [Sustituciones]
= [2,3,5,7,9]
```

- Sustituciones:
  - los `(:)` por `(:)`
  - la `[]` por `ys`
- Definición de `conc` usando `foldr`

```
conc xs ys = (foldr (:) ys) xs
```

7.4. Función de plegado por la izquierda: `foldl`

Definición de suma de lista con acumuladores

- Definición de `suma` con acumuladores:

```
suma :: [Integer] -> Integer
suma = sumaAux 0
    where sumaAux v []     = v
          sumaAux v (x:xs) = sumaAux (v+x) xs
```

- Cálculo con `suma`:

```
  suma [2,3,7] = sumaAux 0 [2,3,7]
               = sumaAux (0+2) [3,7]
               = sumaAux 2 [3,7]
               = sumaAux (2+3) [7]
               = sumaAux 5 [7]
               = sumaAux (5+7) []
               = sumaAux 12 []
               = 12
```

Patrón de definición de recursión con acumulador

- Patrón de definición (generalización de `sumaAux`):

```
f v []     = v
f v (x:xs) = f (v*x) xs
```

- Definición con el patrón `foldl`:

```
suma    = foldl (+) 0
product = foldl (*) 1
or      = foldl (||) False
and     = foldl (&&) True
```

Definición de `foldl`

- Definición de `foldl`:

```
Prelude
foldl :: (a -> b -> a) -> a -> [b ] -> a
foldl f v []     =  v
foldl f v (x:xs) =  foldl f (f v x ) xs
```

7.5. Composición de funciones

Composición de funciones

- Definición de la composición de dos funciones:

```
Prelude
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g  = \x -> f (g x)
```

- Uso de composición para simplificar definiciones:
  - Definiciones sin composición:

```
par n                 = not (impar n)
doVeces f x           = f (f x )
sumaCuadradosPares ns = sum (map (^2) (filter even ns))
```

  - Definiciones con composición:

```
par                 = not . impar
dosVeces f          = f . f
sumaCuadradosPares = sum . map (^2) . filter even
```

Composición de una lista de funciones

- La función identidad:

```
Prelude
id :: a -> a
id =  \x -> x
```

- `(composicionLista fs)` es la composición de la lista de funciones `fs`. Por ejemplo,

```
composicionLista [(*2),(^2)] 3       ~>  18
composicionLista [(^2),(*2)] 3       ~>  36
composicionLista [(/9),(^2),(*2)] 3  ~>  4.0
```

```
composicionLista :: [a -> a] -> (a -> a)
composicionLista =  foldr (.) id
```

7.6. Caso de estudio: Codificación binaria y transmisión de cadenas

- Objetivos:
  1. Definir una función que convierta una cadena en una lista de ceros y unos junto con otra función que realice la conversión opuesta.
  2. Simular la transmisión de cadenas mediante ceros y unos.
- Los números binarios se representan mediante listas de bits en orden inverso. Un bit es cero o uno. Por ejemplo, el número 1101 se representa por [1,0,1,1].
- El tipo `Bit` es el de los bits.

```
type Bit = Int
```

7.6.1. Cambio de bases

Cambio de bases: De binario a decimal

- (bin2int x) es el número decimal correspondiente al número binario x. Por ejemplo,

```
bin2int [1,0,1,1]  ⇝  13
```

El cálculo es

```
  bin2int [1,0,1,1]
= bin2int 1:(0:(1:(1:[])))
=         1+2*(0+2*(1+2*(1+2*0)))
= 13
```

```
bin2int :: [Bit] -> Int
bin2int =  foldr (\x y -> x + 2*y) 0
```

Cambio de base: De decimal a binario

- (int2bin x) es el número binario correspondiente al número decimal x. Por ejemplo,

```
int2bin 13  ⇝  [1,0,1,1]
```

```
int2bin :: Int -> [Bit]
int2bin n | n < 2     = [n]
          | otherwise = n `mod` 2 : int2bin (n `div` 2)
```

Por ejemplo,

```
  int2bin 13
= 13 `mod` 2 : int2bin (13 `div` 2)
= 1 : int2bin (6 `div` 2)
= 1 : (6 `mod` 2 : int2bin (6 `div` 2))
= 1 : (0 : int2bin 3)
= 1 : (0 : (3 `mod` 2 : int2bin (3 `div` 2)))
= 1 : (0 : (1 : int2bin 1))
= 1 : (0 : (1 : (1 : int2bin 0)))
= 1 : (0 : (1 : (1 : [])))
= [1,0,1,1]
```

Cambio de base: Comprobación de propiedades

- Propiedad: Al pasar un número natural a binario con `int2bin` y el resultado a decimal con `bin2int` se obtiene el número inicial.

```
prop_int_bin :: Int -> Bool
prop_int_bin x =
    bin2int (int2bin y) == y
    where y = abs x
```

- Comprobación:

```
*Main> quickCheck prop_int_bin
+++ OK, passed 100 tests.
```

7.6.2. Codificación y descodificación

Creación de octetos

- Un octeto es un grupo de ocho bits.
- `(creaOcteto bs)` es el octeto correspondiente a la lista de bits `bs`; es decir, los 8 primeros elementos de `bs` si su longitud es mayor o igual que 8 y la lista de 8 elemento añadiendo ceros al final de `bs` en caso contrario. Por ejemplo,

```
Main*> creaOcteto [1,0,1,1,0,0,1,1,1,0,0,0]
[1,0,1,1,0,0,1,1]
Main*> creaOcteto [1,0,1,1]
[1,0,1,1,0,0,0,0]
```

```
creaOcteto :: [Bit] -> [Bit]
creaOcteto bs =  take 8 (bs ++ repeat 0)
```

donde `(repeat x)` es una lista infinita cuyo único elemento es `x`.

Codificación

- `(codifica c)` es la codificación de la cadena `c` como una lista de bits obtenida convirtiendo cada carácter en un número Unicode, convirtiendo cada uno de dichos números en un octeto y concatenando los octetos para obtener una lista de bits. Por ejemplo,

```
*Main> codifica "abc"
[1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
```

```
codifica :: String -> [Bit]
codifica =  concat . map (creaOcteto . int2bin . ord)
```

donde (`concat xss`) es la lista obtenida concatenando la lista de listas `xss`.

Codificación

- Ejemplo de codificación,

```
  codifica "abc"
= concat . map (creaOcteto . int2bin . ord) "abc"
= concat . map (creaOcteto . int2bin . ord) ['a','b','c']
= concat [creaOcteto . int2bin . ord 'a',
          creaOcteto . int2bin . ord 'b',
          creaOcteto . int2bin . ord 'c']
= concat [creaOcteto [1,0,0,0,0,1,1],
          creaOcteto [0,1,0,0,0,1,1],
          creaOcteto [1,1,0,0,0,1,1]]
= concat [[1,0,0,0,0,1,1,0],
          [0,1,0,0,0,1,1,0],
          [1,1,0,0,0,1,1,0]]
= [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
```

Separación de octetos

- (`separaOctetos bs`) es la lista obtenida separando la lista de bits `bs` en listas de 8 elementos. Por ejemplo,

```
*Main> separaOctetos [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0]
[[1,0,0,0,0,1,1,0],[0,1,0,0,0,1,1,0]]
```

```
separaOctetos :: [Bit] -> [[Bit]]
separaOctetos [] = []
separaOctetos bs =
    take 8 bs : separaOctetos (drop 8 bs)
```

Descodificación

- (descodifica bs) es la cadena correspondiente a la lista de bits bs. Por ejemplo,

```
*Main> descodifica [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
"abc"
```

```
descodifica :: [Bit] -> String
descodifica =  map (chr . bin2int) . separaOctetos
```

Por ejemplo,

```
  descodifica [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
= (map (chr . bin2int) . separaOctetos)
  [1,0,0,0,0,1,1,0,0,1,0,0,0,1,1,0,1,1,0,0,0,1,1,0]
= map (chr . bin2int) [[1,0,0,0,0,1,1,0],[0,1,0,0,0,1,1,0],[1,1,0,0,0,1,1,0]]
= [(chr . bin2int) [1,0,0,0,0,1,1,0],
   (chr . bin2int) [0,1,0,0,0,1,1,0],
   (chr . bin2int) [1,1,0,0,0,1,1,0]]
= [chr 97, chr 98, chr 99]
= "abc"
```

Transmisión

- Los canales de transmisión pueden representarse mediante funciones que transforman cadenas de bits en cadenas de bits.
- (transmite c t) es la cadena obtenida transmitiendo la cadena t a través del canal c. Por ejemplo,

```
*Main> transmite id "Texto por canal correcto"
"Texto por canal correcto"
```

```
transmite :: ([Bit] -> [Bit]) -> String -> String
transmite canal =  descodifica . canal . codifica
```

Corrección de la transmisión

- Propiedad: Al trasmitir cualquier cadena por el canal identidad se obtiene la cadena.

```
prop_transmite :: String -> Bool
prop_transmite cs =
    transmite id cs == cs
```

- Comprobación de la corrección:

```
*Main> quickCheck prop_transmite
+++ OK, passed 100 tests.
```

Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
    - Cap. 4: Listas.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
    - Cap. 7: Higher-order functions.
3. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
    - Cap. 8: Funciones de orden superior y polimorfismo.
4. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
    - Cap. 9: Generalization: patterns of computation.
    - Cap. 10: Functions as values.

Tema 8

Razonamiento sobre programas

8.1. Razonamiento ecuacional

8.1.1. Cálculo con `longitud`

- Programa:

```
longitud []     = 0                   -- longitud.1
longitud (_:xs) = 1 + longitud xs     -- longitud.2
```

- Propiedad: `longitud [2,3,1] = 3`

- Demostración:
  `longitud [2,3,1]`
  = 1 + `longitud [2,3]` [por `longitud.2`]
  = 1 + (1 + `longitud [3]`) [por `longitud.2`]
  = 1 + (1 + (1 + `longitud []`)) [por `longitud.2`]
  = 1 + (1 + (1 + 0) [por `longitud.1`]
  = 3

8.1.2. Propiedad de `intercambia`

- Programa:

```
intercambia :: (a,b) -> (b,a)
intercambia (x,y) = (y,x)       -- intercambia
```

- Propiedad: `intercambia (intercambia (x,y)) = (x,y)`.

- Demostración:
  `intercambia (intercambia (x,y))`
  `= intercambia (y,x)` [por `intercambia`]
  `= (x,y)` [por `intercambia`]

Comprobación con QuickCheck

- Propiedad:

```
prop_intercambia :: Eq a => a -> a -> Bool
prop_intercambia x y =
    intercambia (intercambia (x,y)) == (x,y)
```

- Comprobación:

```
*Main> quickCheck prop_intercambia
+++ OK, passed 100 tests.
```

8.1.3. Inversa de listas unitarias

- Inversa de una lista:

```
inversa :: [a] -> [a]
inversa []     = []                  -- inversa.1
inversa (x:xs) = inversa xs ++ [x]   -- inversa.2
```

- Prop.: `inversa [x] = [x]`
  `inversa [x]`
  `= inversa (x:[])` [notación de lista]
  `= (inversa []) ++ [x]` [inversa.2]
  `= [] ++ [x]` [inversa.1]
  `= [x]` [def. de ++]

Comprobación con QuickCheck

- Propiedad:

```
prop_inversa_unitaria :: Eq a => a -> Bool
prop_inversa_unitaria x =
    inversa [x] == [x]
```

- Comprobación:

```
*Main> quickCheck prop_inversa_unitaria
+++ OK, passed 100 tests.
```

8.1.4. Razonamiento ecuacional con análisis de casos

- Negación lógica:

Prelude
```
not :: Bool -> Bool
not False = True
not True  = False
```

- Prop.: `not (not x) = x`
- Demostración por casos:
  - Caso 1: `x = True`:
    ```
    not (not True)  = not False  [not.2]
                    = True       [not.1]
    ```
  - Caso 2: `x = False`:
    ```
    not (not False)  = not True  [not.1]
                     = False     [not.2]
    ```

Comprobación con QuickCheck

- Propiedad:

```
prop_doble_negacion :: Bool -> Bool
prop_doble_negacion x =
  not (not x) == x
```

- Comprobación:

```
*Main> quickCheck prop_doble_negacion
+++ OK, passed 100 tests.
```

8.2. Razonamiento por inducción sobre los naturales

8.2.1. Esquema de inducción sobre los naturales

Para demostrar que todos los números naturales tienen una propiedad `P` basta probar:

1. Caso base `n=0`:
   `P(0)`.

2. Caso inductivo `n=(m+1)`:
   Suponiendo `P(m)` demostrar `P(m+1)`.

En el caso inductivo, la propiedad `P(n)` se llama la hipótesis de inducción.

8.2.2. Ejemplo de inducción sobre los naturales

Ejemplo de inducción sobre los naturales: Propiedad

- `(replicate n x)` es la lista formda por `n` elementos iguales a `x`. Por ejemplo,

```
replicate 3 5  ⇝  [5,5,5]
```

Prelude

```
replicate :: Int -> a -> [a]
replicate 0     _ = []
replicate (n+1) x = x : replicate n x
```

- Prop.: `length (replicate n x) = n`

Ejemplo de inducción sobre los naturales: Demostración

- Caso base (`n=0`):
  `length (replicate 0 x)`
  = `length []` [por `replicate.1`]
  = 0 [por def. `length`]

- Caso inductivo (`n=m+1`):
  `length (replicate (m+1) x)`
  = `length (x:(replicate m x))` [por `replicate.2`]
  = `1 + length (replicate m x)` [por def. `length`]
  = `1 + m` [por hip. ind.]
  = `m + 1` [por conmutativa de +]

Ejemplo de inducción sobre los naturales: Verificación

Verificación con QuickCheck:

- Especificación de la propiedad:

```
prop_length_replicate :: Int -> Int -> Bool
prop_length_replicate n xs =
    length (replicate m xs) == m
    where m = abs n
```

- Comprobación de la propiedad:

```
*Main> quickCheck prop_length_replicate
OK, passed 100 tests.
```

8.3. Razonamiento por inducción sobre listas

8.3.1. Esquema de inducción sobre listas

Para demostrar que todas las listas finitas tienen una propiedad `P` basta probar:

1. Caso base `xs=[]`:
   `P([])`.
2. Caso inductivo `xs=(y:ys)`:
   Suponiendo `P(ys)` demostrar `P(y:ys)`.

En el caso inductivo, la propiedad `P(ys)` se llama la hipótesis de inducción.

8.3.2. Asociatividad de ++

- Programa:

```
                          Prelude
(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys              -- ++.1
(x:xs) ++ ys = x : (xs ++ ys) -- ++.2
```

- Propiedad: `xs++(ys++zs)=(xs++ys)++zs`
- Comprobación con QuickCheck:

```
prop_asociativa_conc :: [Int] -> [Int] -> [Int] -> Bool
prop_asociativa_conc xs ys zs =
    xs++(ys++zs)==(xs++ys)++zs
```

```
Main> quickCheck prop_asociatividad_conc
OK, passed 100 tests.
```

- Demostración por inducción en `xs`:

- Caso base `xs=[]`: Reduciendo el lado izquierdo

  ```
  xs++(ys++zs)
  = []++(ys++zs)        [por hipótesis]
  = ys++zs              [por ++.1]
  ```

  y reduciendo el lado derecho

  ```
  (xs++ys)++zs
  = ([]++ys)++zs        [por hipótesis]
  = ys++zs              [por ++.1]
  ```

  Luego, `xs++(ys++zs)=(xs++ys)++zs`

- Demostración por inducción en `xs`:

  - Caso inductivo `xs=a:as`: Suponiendo la hipótesis de inducción `as++(ys++zs)=(as++ys)++zs` hay que demostrar que `(a:as)++(ys++zs)=((a:as)++ys)++zs`

  ```
  (a:as)++(ys++zs)
  = a:(as++(ys++zs))        [por ++.2]
  = a:((as++ys)++zs)        [por hip. ind.]
  = (a:(as++ys))++zs        [por ++.2]
  = ((a:as)++ys)++zs        [por ++.2]
  ```

8.3.3. `[]` es la identidad para ++ por la derecha

- Propiedad: `xs++[]=xs`
- Comprobación con QuickCheck:

```
prop_identidad_concatenacion :: [Int] -> Bool
prop_identidad_concatenacion xs = xs++[] == xs
```

```
Main> quickCheck prop_identidad_concatenacion
OK, passed 100 tests.
```

- Demostración por inducción en `xs`:

  - Caso base `xs=[]`:

  ```
  xs++[]
  = []++[]
  = []          [por ++.1]
  ```

- Caso inductivo `xs=(a:as)`: Suponiendo la hipótesis de inducción `as++[]=as` hay que demostrar que `(a:as)++[]=(a:as)`

  `(a:as)++[]`
  `= a:(as++[])` [por `++.2`]
  `= a:as` [por hip. ind.]

8.3.4. Relación entre `length` y `++`

- Programas:

```
length :: [a] -> Int
length []     = 0                    -- length.1
length (x:xs) = 1 + n_length xs      -- length.2

(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys                    -- ++.1
(x:xs) ++ ys = x : (xs ++ ys)        -- ++.2
```

- Propiedad: `length(xs++ys)=(length xs)+(length ys)`

- Comprobación con QuickCheck:

```
prop_length_append :: [Int] -> [Int] -> Bool
prop_length_append xs ys =
   length(xs++ys)==(length xs)+(length ys)
```

```
Main> quickCheck prop_length_append
OK, passed 100 tests.
```

- Demostración por inducción en `xs`:

  - Caso base `xs=[]`:
    `length([]++ys)`
    `= length ys` [por `++.1`]
    `= 0+(length ys)` [por aritmética]
    `= (length [])+(length ys)` [por `length.1`]

- Demostración por inducción en `xs`:

- Caso inductivo `xs=(a:as)`: Suponiendo la hipótesis de inducción
  `length(as++ys) = (length as)+(length ys)`
  hay que demostrar que
  `length((a:as)++ys) = (length (a:as))+(length ys)`

```
length((a:as)++ys)
= length(a:(as++ys))                     [por ++.2]
= 1 + length(as++ys)                     [por length.2]
= 1 + ((length as) + (length ys))        [por hip. ind.]
= (1 + (length as)) + (length ys)        [por aritmética]
= (length (a:as)) + (length ys)          [por length.2]
```

8.3.5. Relación entre `take` y `drop`

- Programas:

```
take :: Int -> [a] -> [a]
take 0 _        = []                  -- take.1
take _ []       = []                  -- take.2
take n (x:xs)   = x : take (n-1) xs   -- take.3

drop :: Int -> [a] -> [a]
drop 0 xs       = xs                  -- drop.1
drop _ []       = []                  -- drop,2
drop n (_:xs)   = drop (n-1) xs       -- drop.3

(++) :: [a] -> [a] -> [a]
[]     ++ ys = ys                     -- ++.1
(x:xs) ++ ys = x : (xs ++ ys)         -- ++.2
```

- Propiedad: `take n xs ++ drop n xs = xs`
- Comprobación con QuickCheck:

```
prop_take_drop :: Int -> [Int] -> Property
prop_take_drop n xs =
    n >= 0 ==> take n xs ++ drop n xs == xs
```

```
Main> quickCheck prop_take_drop
OK, passed 100 tests.
```

- Demostración por inducción en `n`:

- Caso base n=0:
  take 0 xs ++ drop 0 xs
  = [] ++ xs [por take.1 y drop.1]
  = xs [por ++.1]
- Caso inductivo n=m+1: Suponiendo la hipótesis de inducción 1
  $(\forall xs :: [a])$take m xs ++ drop m xs = xs
  hay que demostrar que
  $(\forall xs :: [a])$take (m+1) xs ++ drop (m+1) xs = xs

Lo demostraremos por inducción en xs:

- Caso base xs=[]:
  take (m+1) [] ++ drop (m+1) []
  = [] ++ [] [por take.2 y drop.2]
  = [] [por ++.1]
- Caso inductivo xs=(a:as): Suponiendo la hip. de inducción 2
  take (m+1) as ++ drop (m+1) as = as
  hay que demostrar que
  take (m+1) (a:as) ++ drop (m+1) (a:as) = (a:as)
  take (m+1) (a:as) ++ drop (m+1) (a:as)
  = (a:(take m as)) ++ (drop m as) [take.3 y drop.3]
  = (a:((take m as) ++ (drop m as)) [por ++.2]
  = a:as [por hip. de ind. 1]

8.3.6. La concatenación de listas vacías es vacía

- Programas:

Prelude
```
null :: [a] -> Bool
null []          = True            -- null.1
null (_:_)       = False           -- null.2

(++) :: [a] -> [a] -> [a]
[]     ++ ys     = ys              -- (++).1
(x:xs) ++ ys     = x : (xs ++ ys)  -- (++).2
```

- Propiedad: null xs = null (xs ++ xs).
- Demostración por inducción en xs:

- Caso 1: `xs = []`: Reduciendo el lado izquierdo

```
null xs
= null []        [por hipótesis]
= True           [por null.1]
```

  y reduciendo el lado derecho

```
null (xs ++ xs)
= null ([] ++ [])        [por hipótesis]
= null []                [por (++).1]
= True                   [por null.1]
```

  Luego, `null xs = null (xs ++ xs)`.

- Demostración por inducción en `xs`:

  - Caso `xs = (y:ys)`: Reduciendo el lado izquierdo

```
null xs
= null (y:ys)        [por hipótesis]
= False              [por null.2
```

  y reduciendo el lado derecho

```
null (xs ++ xs)
= null ((y:ys) ++ (y:ys))        [por hipótesis]
= null (y:(ys ++ (y:ys))         [por (++).2]
= False                          [por null.2
```

  Luego, `null xs = null (xs ++ xs)`.

8.4. Equivalencia de funciones

- Programas:

```
inversa1, inversa2 :: [a] -> [a]
inversa1 []     = []                                      -- inversa1.1
inversa1 (x:xs) = inversa1 xs ++ [x]                      -- inversa1.2

inversa2 xs = inversa2Aux xs []                           -- inversa2.1
    where inversa2Aux []     ys = ys                      -- inversa2Aux.1
          inversa2Aux (x:xs) ys = inversa2Aux xs (x:ys)   -- inversa2Aux.2
```

- Propiedad: `inversa1 xs = inversa2 xs`
- Comprobación con QuickCheck:

```
prop_equiv_inversa :: [Int] -> Bool
prop_equiv_inversa xs = inversa1 xs == inversa2 xs
```

- Demostración: Es consecuencia del siguiente lema:
  `inversa1 xs ++ ys = inversa2Aux xs ys`

  En efecto,

`inversa1 xs`	
= `inversa1 xs ++ []`	[por identidad de ++]
= `inversa2Aux xs ++ []`	[por el lema]
= `inversa2 xs`	[por el `inversa2.1`]

- Demostración del lema: Por inducción en `xs`:

  - Caso base `xs=[]`:

`inversa1 [] ++ ys`	
= `[] ++ ys`	[por `inversa1.1`]
= `ys`	[por `++.1`]
= `inversa2Aux [] ys`	[por `inversa2Aux.1`]

  - Caso inductivo `xs=(a:as)`: La hipótesis de inducción es
    $(\forall ys :: [a])$`inversa1 as ++ ys = inversa2Aux as ys`

    Por tanto,

`inversa1 (a:as) ++ ys`	
= `(inversa1 as ++ [a]) ++ ys`	[por `inversa1.2`]
= `(inversa1 as) ++ ([a] ++ ys)`	[por asociativa de ++]
= `(inversa1 as) ++ (a:ys)`	[por ley unitaria]
= `(inversa2Aux as (a:ys)`	[por hip. de inducción]
= `inversa2Aux (a:as) ys`	[por `inversa2Aux.2`]

8.5. Propiedades de funciones de orden superior

Relación entre `sum` y `map`

- La función `sum`:

Prelude
```
sum :: [Int] -> Int
sum []     = 0
sum (x:xs) = x + sum xs
```

- Propiedad: `sum (map (2*) xs) = 2 * sum xs`
- Comprobación con QuickCheck:

```
prop_sum_map :: [Int] -> Bool
prop_sum_map xs = sum (map (2*) xs) == 2 * sum xs
```

```
*Main> quickCheck prop_sum_map
+++ OK, passed 100 tests.
```

Demostración de la propiedad por inducción en `xs`

- Caso `[]`:

`sum (map (2*) xs)`	
= `sum (map (2*) [])`	[por hipótesis]
= `sum []`	[por `map.1`]
= `0`	[por `sum.1`]
= `2 * 0`	[por aritmética]
= `2 * sum []`	[por `sum.1`]
= `2 * sum xs`	[por hipótesis]

- Caso `xs=(y:ys)`: Entonces,

`sum (map (2*) xs)`	
= `sum (map (2*) (y:ys))`	[por hipótesis]
= `sum ((2*) y : (map (2*) ys))`	[por `map.2`]
= `(2*) y + (sum (map (2*) ys))`	[por `sum.2`]
= `(2*) y + (2 * sum ys)`	[por hip. de inducción]
= `(2 * y) + (2 * sum ys)`	[por `(2*)`]
= `2 * (y + sum ys)`	[por aritmética]
= `2 * sum (y:ys)`	[por `sum.2`]
= `2 * sum xs`	[por hipótesis]

Comprobación de propiedades con argumentos funcionales

- La aplicación de una función a los elemntos de una lista conserva su longitud:

```
prop_map_length (Function _ f) xs =
    length (map f xs) == length xs
```

- En el inicio del fichero hay que escribir

```
import Test.QuickCheck.Function
```

- Comprobación

```
*Main> quickCheck prop_map_length
+++ OK, passed 100 tests.
```

Bibliografía

1. H. C. Cunningham (2007) *Notes on Functional Programming with Haskell*.
2. J. Fokker (1996) *Programación funcional*.
3. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
   - Cap. 13: Reasoning about programs.
4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
   - Cap. 6: Programación con listas.
5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
   - Cap. 8: Reasoning about programs.
6. E.P. Wentworth (1994) *Introduction to Funcional Programming*.

Tema 9

Declaraciones de tipos y clases

9.1. Declaraciones de tipos

Declaraciones de tipos como sinónimos

- Se puede definir un nuevo nombre para un tipo existente mediante una **declaración de tipo**.
- Ejemplo: Las cadenas son listas de caracteres.

Prelude
```
type String = [Char]
```

- El nombre del tipo tiene que empezar por mayúscula.

Declaraciones de tipos nuevos

- Las declaraciones de tipos pueden usarse para facilitar la lectura de tipos. Por ejemplo,
  - Las posiciones son pares de enteros.
    ```
    type Pos = (Int,Int)
    ```
  - `origen` es la posición (0,0).
    ```
    origen :: Pos
    origen = (0,0)
    ```
  - `(izquierda p)` es la posición a la izquierda de la posición `p`. Por ejemplo,
    ```
    izquierda (3,5)  ⇝  (2,5)
    ```
    ```
    izquierda :: Pos -> Pos
    izquierda (x,y) = (x-1,y)
    ```

Declaraciones de tipos parametrizadas

- Las declaraciones de tipos pueden tener parámetros. Por ejemplo,
  - `Par a` es el tipo de pares de elementos de tipo `a`

    ```
    type Par a = (a,a)
    ```

  - `(multiplica p)` es el producto del par de enteros `p`. Por ejemplo,

    ```
    multiplica (2,5)  ⇝  10
    ```

    ```
    multiplica :: Par Int -> Int
    multiplica (x,y) = x*y
    ```

  - `(copia x)` es el par formado con dos copias de `x`. Por ejemplo,

    ```
    copia 5  ⇝  (5,5)
    ```

    ```
    copia :: a -> Par a
    copia x = (x,x)
    ```

Declaraciones anidadas de tipos

- Las declaraciones de tipos pueden anidarse. Por ejemplo,
  - Las posiciones son pares de enteros.

    ```
    type Pos = (Int,Int)
    ```

  - Los movimientos son funciones que va de una posición a otra.

    ```
    type Movimiento = Pos -> Pos
    ```

- Las declaraciones de tipo no pueden ser recursivas. Por ejemplo, el siguiente código es erróneo.

  ```
  type Arbol = (Int,[Arbol])
  ```

  Al intentar cargarlo da el mensaje de error

  ```
  Cycle in type synonym declarations
  ```

9.2. Definiciones de tipos de datos

Definición de tipos con `data`

- En Haskell pueden definirse nuevos tipos mediante `data`.
- El tipo de los booleanos está formado por dos valores para representar lo falso y lo verdadero.

Prelude
```
data Bool = False | True
```

- El símbolo `|` se lee como "o".
- Los valores `False` y `True` se llaman los **constructores** del tipo `Bool`.
- Los nombres de los constructores tienen que empezar por mayúscula.

Uso de los valores de los tipos definidos

- Los valores de los tipos definidos pueden usarse como los de los predefinidos.
- Definición del tipo de movimientos:

```
data Mov = Izquierda | Derecha | Arriba | Abajo
```

- Uso como argumento: `(movimiento m p)` es la posición obtenida aplicando el movimiento `m` a la posición `p`. Por ejemplo,

```
movimiento Arriba (2,5)  ⇝ (2,6)
```

```
movimiento :: Mov -> Pos -> Pos
movimiento Izquierda (x,y) = (x-1,y)
movimiento Derecha   (x,y) = (x+1,y)
movimiento Arriba    (x,y) = (x,y+1)
movimiento Abajo     (x,y) = (x,y-1)
```

- Uso en listas: `(movimientos ms p)` es la posición obtenida aplicando la lista de movimientos `ms` a la posición `p`. Por ejemplo,

```
movimientos [Arriba, Izquierda] (2,5)  ⇝  (1,6)
```

```
movimientos :: [Mov] -> Pos -> Pos
movimientos []     p = p
movimientos (m:ms) p = movimientos ms (movimiento m p)
```

- Uso como valor: (opuesto m) es el movimiento opuesto de m.

```
movimiento (opuesto Arriba) (2,5)  ⇝ (2,4)
```

```
opuesto :: Mov -> Mov
opuesto Izquierda = Derecha
opuesto Derecha   = Izquierda
opuesto Arriba    = Abajo
opuesto Abajo     = Arriba
```

Definición de tipo con constructores con parámetros

- Los constructores en las definiciones de tipos pueden tener parámetros.
- Ejemplo de definición

```
data Figura = Circulo Float | Rect Float Float
```

- Tipos de los constructores:

```
*Main> :type Circulo
Circulo :: Float -> Figura
*Main> :type Rect
Rect :: Float -> Float -> Figura
```

- Uso del tipo como valor: (cuadrado n) es el cuadrado de lado n.

```
cuadrado :: Float -> Figura
cuadrado n = Rect n n
```

- Uso del tipo como argumento: (area f) es el área de la figura f. Por ejemplo,

```
area (Circulo 1)   ⇝  3.1415927
area (Circulo 2)   ⇝  12.566371
area (Rect 2 5)    ⇝  10.0
area (cuadrado 3)  ⇝  9.0
```

```
area :: Figura -> Float
area (Circulo r) = pi*r^2
area (Rect x y)  = x*y
```

Definición de tipos con parámetros

- Los tipos definidos pueden tener parámetros.
- Ejemplo de tipo con parámetro

```
Prelude
data Maybe a = Nothing | Just a
```

- `(divisionSegura m n)` es la división de `m` entre `n` si `n` no es cero y nada en caso contrario. Por ejemplo,

```
divisionSegura 6 3  ⇝  Just 2
divisionSegura 6 0  ⇝  Nothing
```

```
divisionSegura :: Int -> Int -> Maybe Int
divisionSegura _ 0 = Nothing
divisionSegura m n = Just (m `div` n)
```

- `(headSegura xs)` es la cabeza de `xs` si `xs` es no vacía y nada en caso contrario. Por ejemplo,

```
headSegura [2,3,5]  ⇝  Just 2
headSegura []       ⇝  Nothing
```

```
headSegura :: [a] -> Maybe a
headSegura [] = Nothing
headSegura xs = Just (head xs)
```

9.3. Definición de tipos recursivos

Definición de tipos recursivos: Los naturales

- Los tipos definidos con `data` pueden ser recursivos.
- Los naturales se construyen con el cero y la función sucesor.

```
data Nat = Cero | Suc Nat
           deriving Show
```

- Tipos de los constructores:

```
*Main> :type Cero
Cero :: Nat
*Main> :type Suc
Suc :: Nat -> Nat
```

- Ejemplos de naturales:

```
Cero
Suc Cero
Suc (Suc Cero)
Suc (Suc (Suc Cero))
```

Definiciones con tipos recursivos

- `(nat2int n)` es el número entero correspondiente al número natural `n`. Por ejemplo,

```
nat2int (Suc (Suc (Suc Cero)))  ⇝  3
```

```
nat2int :: Nat -> Int
nat2int Cero    = 0
nat2int (Suc n) = 1 + nat2int n
```

- `(int2nat n)` es el número natural correspondiente al número entero `n`. Por ejemplo,

```
int2nat 3  ⇝  Suc (Suc (Suc Cero))
```

```
int2nat :: Int -> Nat
int2nat 0     = Cero
int2nat (n+1) = Suc (int2nat n)
```

- `(suma m n)` es la suma de los número naturales `m` y `n`. Por ejemplo,

```
*Main> suma (Suc (Suc Cero)) (Suc Cero)
Suc (Suc (Suc Cero))
```

```
suma :: Nat -> Nat -> Nat
suma Cero    n = n
suma (Suc m) n = Suc (suma m n)
```

- Ejemplo de cálculo:

```
  suma (Suc (Suc Cero)) (Suc Cero)
= Suc (suma (Suc Cero) (Suc Cero))
= Suc (Suc (suma Cero (Suc Cero)))
= Suc (Suc (Suc Cero))
```

Tipo recursivo con parámetro: Las listas

- Definicón del tipo lista:

```
data Lista a = Nil | Cons a (Lista a)
```

- (`longitud xs`) es la longitud de la lista `xs`. Por ejemplo,

```
longitud (Cons 2 (Cons 3 (Cons 5 Nil)))  ⇝  3
```

```
longitud :: Lista a -> Int
longitud Nil           = 0
longitud (Cons _ xs) = 1 + longitud xs
```

Definición de tipos recursivos: Los árboles binarios

- Ejemplo de árbol binario:

```
     5
    / \
   /   \
  3     7
 / \   / \
1   4 6   9
```

- Definición del tipo de árboles binarios:

```
data Arbol = Hoja Int | Nodo Arbol Int Arbol
```

- Representación del ejemplo

```
ejArbol = Nodo (Nodo (Hoja 1) 3 (Hoja 4))
               5
               (Nodo (Hoja 6) 7 (Hoja 9))
```

Definiciones sobre árboles binarios

- `(ocurre m a)` se verifica si `m` ocurre en el árbol `a`. Por ejemplo,

```
ocurre 4  ejArbol  ⇝  True
ocurre 10 ejArbol  ⇝  False
```

```
ocurre :: Int -> Arbol -> Bool
ocurre m (Hoja n)     = m == n
ocurre m (Nodo i n d) = m == n || ocurre m i || ocurre m d
```

- `(aplana a)` es la lista obtenida aplanando el árbol `a`. Por ejemplo,

```
aplana ejArbol  ⇝  [1,3,4,5,6,7,9]
```

```
aplana :: Arbol -> [Int]
aplana (Hoja n)     = [n]
aplana (Nodo i n d) = aplana i ++ [n] ++ aplana d
```

Definiciones sobre árboles binarios

- Un árbol es ordenado si el valor de cada nodo es mayor que los de su subárbol izquierdo y menor que los de su subárbol derecho.
- El árbol del ejemplo es ordenado.
- `(ocurreEnArbolOrdenado m a)` se verifica si `m` ocurre en el árbol ordenado `a`. Por ejemplo,

```
ocurreEnArbolOrdenado 4  ejArbol  ⇝  True
ocurreEnArbolOrdenado 10 ejArbol  ⇝  False
```

```
ocurreEnArbolOrdenado :: Int -> Arbol -> Bool
ocurreEnArbolOrdenado m (Hoja n)  =  m == n
ocurreEnArbolOrdenado m (Nodo i n d)
     | m == n      =  True
     | m < n       =  ocurreEnArbolOrdenado m i
     | otherwise   =  ocurreEnArbolOrdenado m d
```

Definiciones de distintos tipos de árboles

- Árboles binarios con valores en las hojas:

```
data Arbol a = Hoja a | Nodo (Arbol a) (Arbol a)
```

- Árboles binarios con valores en los nodos:

```
data Arbol a = Hoja | Nodo (Arbol a) a (Arbol a)
```

- Árboles binarios con valores en las hojas y en los nodos:

```
data Arbol a b = Hoja a | Nodo (Arbol a b) b (Arbol a b)
```

- Árboles con un número variable de sucesores:

```
data Arbol a = Nodo a [Arbol a]
```

9.4. Sistema de decisión de tautologías

Sintaxis de la lógica proposicional

- Definición de fórmula proposicional:
  - Las variables proposicionales son fórmulas.
  - Si F es una fórmula, entonces $\neg F$ también lo es.
  - Si F y G son fórmulas, entonces $F \wedge G$ y $F \to G$ también lo son.
- Tipo de dato de fórmulas proposicionales:

```
data FProp = Const Bool
           | Var Char
           | Neg FProp
           | Conj FProp FProp
           | Impl FProp FProp
           deriving Show
```

- Ejemplos de fórmulas proposicionales:
  1. $A \wedge \neg A$
  2. $(A \wedge B) \to A$

3. $A \to (A \land B)$
4. $(A \to (A \to B)) \to B$

```
p1, p2, p3, p4 :: FProp
p1 = Conj (Var 'A') (Neg (Var 'A'))
p2 = Impl (Conj (Var 'A') (Var 'B')) (Var 'A')
p3 = Impl (Var 'A') (Conj (Var 'A') (Var 'B'))
p4 = Impl (Conj (Var 'A') (Impl (Var 'A') (Var 'B')))
          (Var 'B')
```

Semántica de la lógica proposicional

- Tablas de verdad de las conectivas:

i	$\neg i$
T	F
F	T

i	j	$i \land j$	$i \to j$
T	T	T	T
T	F	F	F
F	T	F	T
F	F	F	T

- Tabla de verdad para $(A \to B) \lor (B \to A)$:

A	B	$(A \to B)$	$(B \to A)$	$(A \to B) \lor (B \to A)$
T	T	T	T	T
T	F	F	T	T
F	T	T	F	T
F	F	T	T	T

- Las interpretaciones son listas formadas por el nombre de una variable proposicional y un valor de verdad.

```
type Interpretacion = [(Char, Bool)]
```

- `(valor i p)` es el valor de la fórmula `p` en la interpretación `i`. Por ejemplo,

```
valor [('A',False),('B',True)] p3  ⇝  True
valor [('A',True),('B',False)] p3  ⇝  False
```

```
valor :: Interpretacion -> FProp -> Bool
valor _ (Const b)  = b
valor i (Var x)    = busca x i
valor i (Neg p)    = not (valor i p)
valor i (Conj p q) = valor i p && valor i q
valor i (Impl p q) = valor i p <= valor i q
```

- (busca c t) es el valor del primer elemento de la lista de asociación t cuya clave es c. Por ejemplo,

```
busca 2 [(1,'a'),(3,'d'),(2,'c')]  ⇝  'c'
```

```
busca :: Eq c => c -> [(c,v)] -> v
busca c t = head [v | (c',v) <- t, c == c']
```

- (variables p) es la lista de los nombres de las variables de p.

```
variables p3  ⇝  "AAB"
```

```
variables :: FProp -> [Char]
variables (Const _)  = []
variables (Var x)    = [x]
variables (Neg p)    = variables p
variables (Conj p q) = variables p ++ variables q
variables (Impl p q) = variables p ++ variables q
```

- (interpretacionesVar n) es la lista de las interpretaciones con n variables. Por ejemplo,

```
*Main> interpretacionesVar 2
[[False,False],
 [False,True],
 [True,False],
 [True,True]]
```

```
interpretacionesVar :: Int -> [[Bool]]
interpretacionesVar 0     = [[]]
interpretacionesVar (n+1) =
    map (False:) bss ++ map (True:) bss
    where bss = interpretacionesVar n
```

- (interpretaciones p) es la lista de las interpretaciones de la fórmula p. Por ejemplo,

```
*Main> interpretaciones p3
[[('A',False),('B',False)],
 [('A',False),('B',True)],
 [('A',True),('B',False)],
 [('A',True),('B',True)]]
```

```
interpretaciones :: FProp -> [Interpretacion]
interpretaciones p =
    [zip vs i | i <- interpretacionesVar (length vs)]
    where vs = nub (variables p)
```

Decisión de tautología

- `(esTautologia p)` se verifica si la fórmula `p` es una tautología. Por ejemplo,

```
esTautologia p1  ⇝  False
esTautologia p2  ⇝  True
esTautologia p3  ⇝  False
esTautologia p4  ⇝  True
```

```
esTautologia :: FProp -> Bool
esTautologia p =
    and [valor i p | i <- interpretaciones p]
```

9.5. Máquina abstracta de cálculo aritmético

Evaluación de expresiones aritméticas

- Una expresión aritmética es un número entero o la suma de dos expresiones.

```
data Expr =  Num Int | Suma Expr Expr
```

- `(valorEA x)` es el valor de la expresión aritmética `x`.

```
valorEA (Suma (Suma (Num 2) (Num 3)) (Num 4))  ⇝  9
```

```
valorEA :: Expr -> Int
valorEA (Num n)    = n
valorEA (Suma x y) = valorEA x + valorEA y
```

- Cálculo:

```
  valorEA (Suma (Suma (Num 2) (Num 3)) (Num 4))
= (valorEA (Suma (Num 2) (Num 3))) + (valorEA (Num 4))
= (valorEA (Suma (Num 2) (Num 3))) + 4
= (valorEA (Num 2) + (valorEA (Num 3))) + 4
= (2 + 3) + 4
= 9
```

Máquina de cálculo aritmético

- La pila de control de la máquina abstracta es una lista de operaciones.

```
type PControl = [Op]
```

- Las operaciones son meter una expresión en la pila o sumar un número con el primero de la pila.

```
data Op =  METE Expr | SUMA Int
```

- (`eval x p`) evalúa la expresión `x` con la pila de control `p`. Por ejemplo,

```
eval (Suma (Suma (Num 2) (Num 3)) (Num 4)) []  ⇝  9
eval (Suma (Num 2) (Num 3)) [METE (Num 4)]     ⇝  9
eval (Num 3) [SUMA 2, METE (Num 4)]            ⇝  9
eval (Num 4) [SUMA 5]                          ⇝  9
```

```
eval :: Expr -> PControl -> Int
eval (Num n)    p = ejec p n
eval (Suma x y) p = eval x (METE y : p)
```

- (`ejec p n`) ejecuta la lista de control `p` sobre el entero `n`. Por ejemplo,

```
ejec [METE (Num 3), METE (Num 4)] 2  ⇝  9
ejec [SUMA 2, METE (Num 4)]       3  ⇝  9
ejec [METE (Num 4)]               5  ⇝  9
ejec [SUMA 5]                     4  ⇝  9
ejec []                           9  ⇝  9
```

```
ejec :: PControl -> Int -> Int
ejec []           n = n
ejec (METE y : p) n = eval y (SUMA n : p)
ejec (SUMA n : p) m = ejec p (n+m)
```

- (evalua e) evalúa la expresión aritmética e con la máquina abstracta. Por ejemplo,

```
evalua (Suma (Suma (Num 2) (Num 3)) (Num 4))  ⇝  9
```

```
evalua :: Expr -> Int
evalua e = eval e []
```

- Evaluación:

```
  eval (Suma (Suma (Num 2) (Num 3)) (Num 4)) []
= eval (Suma (Num 2) (Num 3)) [METE (Num 4)]
= eval (Num 2) [METE (Num 3), METE (Num 4)]
= ejec [METE (Num 3), METE (Num 4)] 2
= eval (Num 3) [SUMA 2, METE (Num 4)]
= ejec [SUMA 2, METE (Num 4)] 3
= ejec [METE (Num 4)] (2+3)
= ejec [METE (Num 4)] 5
= eval (Num 4) [SUMA 5]
= ejec [SUMA 5] 4
= ejec [] (5+4)
= ejec [] 9
= 9
```

9.6. Declaraciones de clases y de instancias

Declaraciones de clases

- Las clases se declaran mediante el mecanismo class.
- Ejemplo de declaración de clases:

Prelude

```
class Eq a where
    (==), (/=) :: a -> a -> Bool

    -- Minimal complete definition: (==) or (/=)
    x == y = not (x/=y)
    x /= y = not (x==y)
```

Declaraciones de instancias

- Las instancias se declaran mediante el mecanismo `instance`.
- Ejemplo de declaración de instancia:

Prelude
```
instance Eq Bool where
    False == False = True
    True  == True  = True
    _     == _     = False
```

Extensiones de clases

- Las clases pueden extenderse mediante el mecanismo `class`.
- Ejemplo de extensión de clases:

Prelude
```
class (Eq a) => Ord a where
    compare                :: a -> a -> Ordering
    (<), (<=), (>=), (>)   :: a -> a -> Bool
    max, min               :: a -> a -> a

    -- Minimal complete definition: (<=) or compare
    -- using compare can be more efficient for complex types
    compare x y | x==y      = EQ
                | x<=y      = LT
                | otherwise = GT

    x <= y                  = compare x y /= GT
    x <  y                  = compare x y == LT
    x >= y                  = compare x y /= LT
    x >  y                  = compare x y == GT

    max x y   | x <= y      = y
              | otherwise   = x
    min x y   | x <= y      = x
              | otherwise   = y
```

Instancias de clases extendidas

- Las instancias de las clases extendidas pueden declararse mediante el mecanismo `instance`.
- Ejemplo de declaración de instancia:

Prelude
```
instance Ord Bool where
     False <= _     = True
     True  <= True  = True
     True  <= False = False
```

Clases derivadas

- Al definir un nuevo tipo con `data` puede declarse como instancia de clases mediante el mecanismo `deriving`.
- Ejemplo de clases derivadas:

Prelude
```
data Bool = False | True
            deriving (Eq, Ord, Read, Show)
```

- Comprobación:

```
False == False          ⇝  True
False < True            ⇝  True
show False              ⇝  "False"
read "False" :: Bool    ⇝  False
```

- Para derivar un tipo cuyos constructores tienen argumentos como derivado, los tipos de los argumentos tienen que ser instancias de las clases derivadas.
- Ejemplo:

```
data Figura = Circulo Float | Rect Float Float
              deriving (Eq, Ord, Show)
```

se cumple que `Float` es instancia de `Eq`, `Ord` y `Show`.

```
*Main> :info Float
...
instance Eq Float
instance Ord Float
instance Show Float
...
```

Bibliografía

1. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
   - Cap. 10: Declaring types and classes.
2. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
   - Cap. 4: Definición de tipos.
   - Cap. 5: El sistema de clases de Haskell.
3. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
   - Cap. 12: Overloading and type classes.
   - Cap. 13: Checking types.
   - Cap. 14: Algebraic types.

Tema 10

Evaluación perezosa

10.1. Estrategias de evaluación

Estrategias de evaluación

- Para los ejemplos se considera la función

```
mult :: (Int,Int) -> Int
mult (x,y) = x*y
```

- Evaluación mediante paso de parámetros por valor (o por más internos):

	mult (1+2,2+3)	
=	mult (3,5)	[por def. de +]
=	3*5	[por def. de mult]
=	15	[por def. de *]

- Evaluación mediante paso de parámetros por nombre (o por más externos):

	mult (1+2,2+3)	
=	(1+2)*(3+5)	[por def. de mult]
=	3*5	[por def. de +]
=	15	[por def. de *]

Evaluación con lambda expresiones

- Se considera la función

```
mult' :: Int -> Int -> Int
mult' x = \y -> x*y
```

- Evaluación:

	mult′ (1+2) (2+3)	
=	mult′ 3 (2+3)	[por def. de +]
=	(λy → 3*y) (2+3)	[por def. de mult′]
=	(λy → 3*y) 5	[por def. de +]
=	3*5	[por def. de +]
=	15	[por def. de *]

10.2. Terminación

Procesamiento con el infinito

- Definición de infinito

```
inf :: Int
inf = 1 + inf
```

- Evaluación de infinito en Haskell:

```
*Main> inf
  C-c C-cInterrupted.
```

- Evaluación de infinito:

	inf	
=	1 + inf	[por def. inf]
=	1 + (1 + inf)	[por def. inf]
=	1 + (1 + (1 + inf))	[por def. inf]
=	...	

Procesamiento con el infinito

- Evaluación mediante paso de parámetros por valor:

	fst (0,inf)	
=	fst (0,1 + inf)	[por def. inf]
=	fst (0,1 + (1 + inf))	[por def. inf]
=	fst (0,1 + (1 + (1 + inf)))	[por def. inf]
=	...	

- Evaluación mediante paso de parámetros por nombre:

	fst (0,inf)	
=	0	[por def. fst]

- Evaluación Haskell con infinito:

```
*Main> fst (0,inf)
0
```

10.3. Número de reducciones

Número de reducciones según las estrategias

- Para los ejemplos se considera la función

```
cuadrado :: Int -> Int
cuadrado n = n * n
```

- Evaluación mediante paso de parámetros por valor:

	cuadrado (1+2)	
=	cuadrado 3	[por def. +]
=	3*3	[por def. cuadrado]
=	9	[por def. de *]

- Evaluación mediante paso de parámetros por nombre:

	cuadrado (1+2)	
=	(1+2)*(1+2)	[por def. cuadrado]
=	3*(1+2)	[por def. de +]
=	3*3	[por def. de +]
=	9	[por def. de *]

Evaluación perezosa e impaciente

- En la evaluación mediante paso de parámetros por nombre los argumentos pueden evaluarse más veces que en el paso por valor.
- Se puede usar punteros para compartir valores de expresiones.
- La evaluación mediante paso de parámetros por nombre usando punteros para compartir valores de expresiones se llama **evaluación perezosa**.
- La evaluación mediante paso de parámetros por valor se llama **evaluación impaciente**.
- Evaluación perezosa del ejemplo anterior:

	cuadrado (1+2)	
=	x*x con x = 1+2	[por def. cuadrado]
=	3*3	[por def. de +]
=	9	[por def. de *]

- Haskell usa evaluación perezosa.

10.4. Estructuras infinitas

Programación con estructuras infinitas

- `unos` es una lista infinita de unos.

```
unos :: [Int]
unos = 1 : unos
```

- Evaluación:

	unos	
=	1 : unos	[por def. unos]
=	1 : (1 : unos)	[por def. unos]
=	1 : (1 : (1 : unos))	[por def. unos]
=	...	

- Evaluación en Haskell:

```
*Main> unos
[1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,...
```

Evaluación con estructuras infinitas

- Evaluación impaciente:

	head unos	
=	head (1 : unos)	[por def. unos]
=	head (1 : (1 : unos))	[por def. unos]
=	head (1 : (1 : (1 : unos)))	[por def. unos]
=	...	

- Evaluación perezosa:

	head unos	
=	head (1 : unos)	[por def. unos]
=	1	[por def. head]

- Evaluación Haskell:

```
*Main> head unos
1
```

10.5. Programación modular

Programación modular

- La evaluación perezosa permite separar el control de los datos.
- Para los ejemplos se considera la función

Prelude
```
take :: Int -> [a] -> [a]
take n _  | n <= 0  = []
take _ []           = []
take n (x:xs)       = x : take (n-1) xs
```

- Ejemplo de separación del control (tomar 2 elementos) de los datos (una lista infinita de unos):

	take 2 unos	
=	take 2 (1 : unos)	[por def. unos]
=	1 : (take 1 unos)	[por def. take]
=	1 : (take 1 (1 : unos))	[por def. unos]
=	1 : (1 : (take 0 unos))	[por def. take]
=	1 : (1 : [])	[por def. take]
=	[1,1]	[por notación de listas]

Terminación de evaluaciones con estructuras infinitas

- Ejemplo de no terminación:

```
*Main> [1..]
[1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,...
```

- Ejemplo de terminación:

```
*Main> take 3 [1..]
[1,2,3]
```

- Ejemplo de no terminación:

```
*Main> filter (<=3) [1..]
[1,2,3  C-c C-c Interrupted.
```

- Ejemplo de no terminación:

```
*Main> takeWhile (<=3) [1..]
[1,2,3]
```

La criba de Erastótenes

- La criba de Erastótenes

2	3	4	5	6	7	8	9	10	11	12	13	14	15	...
	3		5		7		9		11		13		15	...
			5		7				11		13			...
					7				11		13			...
									11		13			...
											13			...

- Definición

```
primos :: [Int ]
primos = criba [2..]

criba :: [Int] -> [Int]
criba (p:xs) = p : criba [x | x <- xs, x `mod` p /= 0]
```

- Evaluación:

```
take 15 primos ⇝ [2,3,5,7,11,13,17,19,23,29,31,37,41,43,47]
```

- Cálculo:

```
  primos
= criba [2..]
= criba (2 : [3..])
= 2 : (criba [x | x <- [3..], x `mod` 2 /= 0])
= 2 : (criba (3 : [x | x <- [4..], x `mod` 2 /= 0]))
= 2 : 3 : (criba [x | x <- [4..], x `mod` 2 /= 0,
                                  x `mod` 3 /= 0])
= 2 : 3 : (criba (5 : [x | x <- [6..], x `mod` 2 /= 0,
                                       x `mod` 3 /= 0]))
= 2 : 3 : 5 : (criba ([x | x <- [6..], x `mod` 2 /= 0,
                                       x `mod` 3 /= 0,
                                       x `mod` 5 /= 0]))
= ...
```

10.6. Aplicación estricta

Ejemplo de programa sin aplicación estricta

- (sumaNE xs) es la suma de los números de xs. Por ejemplo,

```
sumaNE [2,3,5]  ⇝  10
```

```
sumaNE :: [Int] -> Int
sumaNE xs = sumaNE' 0 xs

sumaNE' :: Int -> [Int] -> Int
sumaNE' v []     = v
sumaNE' v (x:xs) = sumaNE' (v+x) xs
```

- Evaluación: :

	sumaNE [2,3,5]	
=	sumaNE′ 0 [2,3,5]	[por def. sumaNE]
=	sumaNE′ (0+2) [3,5]	[por def. sumaNE′]
=	sumaNE′ ((0+2)+3) [5]	[por def. sumaNE′]
=	sumaNE′ (((0+2)+3)+5) []	[por def. sumaNE′]
=	((0+2)+3)+5	[por def. sumaNE′]
=	(2+3)+5	[por def. +]
=	5+5	[por def. +]
=	10	[por def. +]

Ejemplo de programa con aplicación estricta

- (sumaE xs) es la suma de los números de xs. Por ejemplo,

```
sumaE [2,3,5]  ⇝  10
```

```
sumaE :: [Int] -> Int
sumaE xs = sumaE' 0 xs

sumaE' :: Int -> [Int] -> Int
sumaE' v []     = v
sumaE' v (x:xs) = (sumaE' $! (v+x)) xs
```

- Evaluación: :

	sumaE [2,3,5]	
=	sumaE′ 0 [2,3,5]	[por def. sumaE]
=	(sumaE′ $! (0+2)) [3,5]	[por def. sumaE′]
=	sumaE′ 2 [3,5]	[por aplicación de $!]
=	(sumaE′ $! (2+3)) [5]	[por def. sumaE′]
=	sumaE′ 5 [5]	[por aplicación de $!]
=	(sumaE′ $! (5+5)) []	[por def. sumaE′]
=	sumaE′ 10 []	[por aplicación de $!]
=	10	[por def. sumaE′]

Comparación de consumo de memoria

- Comparación de consumo de memoria:

```
*Main> sumaNE [1..1000000]
*** Exception: stack overflow
*Main> sumaE [1..1000000]
1784293664
*Main> :set +s
*Main> sumaE [1..1000000]
1784293664
(2.16 secs, 145435772 bytes)
```

Plegado estricto

- Versión estricta de `foldl` en el `Data.List`

```
foldl' :: (a -> b -> a) -> a -> [b] -> a
foldl' f a []     = a
foldl' f a (x:xs) = (foldl' f $! f a x) xs
```

- Comparación de plegado y plegado estricto:s

```
*Main> foldl (+) 0 [2,3,5]
10
*Main> foldl' (+) 0 [2,3,5]
10
*Main> foldl (+) 0 [1..1000000]
*** Exception: stack overflow
*Main> foldl' (+) 0 [1..1000000]
500000500000
```

Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
   - Cap. Cap. 7: Eficiencia.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
   - Cap. 12: Lazy evaluation.
3. B. O'Sullivan, D. Stewart y J. Goerzen *Real World Haskell*. O'Reilly, 2008.
   - Cap. 2: Types and Functions.
4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
   - Cap. 2: Introducción a Haskell.
   - Cap. 8: Evaluación perezosa. Redes de procesos.
5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
   - Cap. 17: Lazy programming.

Tema 11

Aplicaciones de programación funcional

11.1. El juego de cifras y letras

11.1.1. Introducción

Presentación del juego

- *Cifras y letras* es un programa de Canal Sur que incluye un juego numérico cuya esencia es la siguiente:

  > Dada una sucesión de números naturales y un número objetivo, intentar construir una expresión cuyo valor es el objetivo combinando los números de la sucesión usando suma, resta, multiplicación, división y paréntesis. Cada número de la sucesión puede usarse como máximo una vez. Además, todos los números, incluyendo los resultados intermedios tienen que ser enteros positivos (1,2,3,...).

- Ejemplos
  - Dada la sucesión 1, 3, 7, 10, 25, 50 y el objetivo 765, una solución es (1+50)*(25−10).
  - Para el problema anterior, existen 780 soluciones.
  - Con la sucesión anterior y el objetivo 831, no hay solución.

Formalización del problema: Operaciones

- Las operaciones son sumar, restar, multiplicar o dividir.

```
data Op = Sum | Res | Mul | Div

instance Show Op where
```

```
    show Sum = "+"
    show Res = "-"
    show Mul = "*"
    show Div = "/"
```

- ops es la lista de las operaciones.

```
ops :: [Op]
ops = [Sum,Res,Mul,Div]
```

Operaciones válidas

- (valida o x y) se verifica si la operación o aplicada a los números naturales x e y da un número natural. Por ejemplo,

```
valida Res 5 3  ⇝  True
valida Res 3 5  ⇝  False
valida Div 6 3  ⇝  True
valida Div 6 4  ⇝  False
```

```
valida :: Op -> Int -> Int -> Bool
valida Sum _ _ = True
valida Res x y = x > y
valida Mul _ _ = True
valida Div x y = y /= 0 && x `mod` y == 0
```

Aplicación de operaciones

- (aplica o x y) es el resultado de aplicar la operación o a los números naturales x e y. Por ejemplo,

```
aplica Sum 2 3  ⇝  5
aplica Div 6 3  ⇝  2
```

```
aplica :: Op -> Int -> Int -> Int
aplica Sum x y = x + y
aplica Res x y = x - y
aplica Mul x y = x * y
aplica Div x y = x `div` y
```

Expresiones

- Las expresiones son números enteros o aplicaciones de operaciones a dos expresiones.

```
data Expr = Num Int | Apl Op Expr Expr

instance Show Expr where
   show (Num n)     = show n
   show (Apl o i d) = parentesis i ++ show o ++ parentesis d
                      where
                         parentesis (Num n) = show n
                         parentesis e       = "(" ++ show e ++ ")"
```

- Ejemplo: Expresión correspondiente a (1+50)*(25−10)

```
ejExpr :: Expr
ejExpr =  Apl Mul e1 e2
    where e1 = Apl Sum (Num 1) (Num 50)
          e2 = Apl Res (Num 25) (Num 10)
```

Números de una expresión

- `(numeros e)` es la lista de los números que aparecen en la expresión `e`. Por ejemplo,

```
*Main> numeros (Apl Mul (Apl Sum (Num 2) (Num 3)) (Num 7))
[2,3,7]
```

```
numeros :: Expr -> [Int]
numeros (Num n)     = [n]
numeros (Apl _ l r) = numeros l ++ numeros r
```

Valor de una expresión

- `(valor e)` es la lista formada por el valor de la expresión `e` si todas las operaciones para calcular el valor de `e` son números positivos y la lista vacía en caso contrario. Por ejemplo,

```
valor (Apl Mul (Apl Sum (Num 2) (Num 3)) (Num 7)) ⇝ [35]
valor (Apl Res (Apl Sum (Num 2) (Num 3)) (Num 7)) ⇝ []
valor (Apl Sum (Apl Res (Num 2) (Num 3)) (Num 7)) ⇝ []
```

```
valor :: Expr -> [Int]
valor (Num n)     = [n | n > 0]
valor (Apl o i d) = [aplica o x y | x <- valor i
                                  , y <- valor d
                                  , valida o x y]
```

Funciones combinatorias: Sublistas

- (sublistas xs) es la lista de las sublistas de xs. Por ejemplo,

```
*Main> sublistas "bc"
["","c","b","bc"]
*Main> sublistas "abc"
["","c","b","bc","a","ac","ab","abc"]
```

```
sublistas :: [a] -> [[a]]
sublistas []     = [[]]
sublistas (x:xs) = yss ++ map (x:) yss
    where yss = sublistas xs
```

Funciones combinatoria: Intercalado

- (intercala x ys) es la lista de las listas obtenidas intercalando x entre los elementos de ys. Por ejemplo,

```
intercala 'x' "bc"   ⇝  ["xbc","bxc","bcx"]
intercala 'x' "abc"  ⇝  ["xabc","axbc","abxc","abcx"]
```

```
intercala :: a -> [a] -> [[a]]
intercala x []     = [[x]]
intercala x (y:ys) =
    (x:y:ys) : map (y:) (intercala x ys)
```

Funciones combinatoria: Permutaciones

- (permutaciones xs) es la lista de las permutaciones de xs. Por ejemplo,

```
*Main> permutaciones "bc"
["bc","cb"]
*Main> permutaciones "abc"
["abc","bac","bca","acb","cab","cba"]
```

```
permutaciones :: [a] -> [[a]]
permutaciones []     = [[]]
permutaciones (x:xs) =
    concat (map (intercala x) (permutaciones xs))
```

Funciones combinatoria: Elecciones

- (`elecciones xs`) es la lista formada por todas las sublistas de `xs` en cualquier orden. Por ejemplo,

```
*Main> elecciones "abc"
["","c","b","bc","cb","a","ac","ca","ab","ba",
 "abc","bac","bca","acb","cab","cba"]
```

```
elecciones :: [a] -> [[a]]
elecciones xs =
    concat (map permutaciones (sublistas xs))
```

Reconocimiento de las soluciones

- (`solucion e ns n`) se verifica si la expresión `e` es una solución para la sucesión `ns` y objetivo `n`; es decir. si los números de `e` es una posible elección de `ns` y el valor de `e` es `n`. Por ejemplo,

```
solucion ejExpr [1,3,7,10,25,50] 765  =>  True
```

```
solucion :: Expr -> [Int] -> Int -> Bool
solucion e ns n =
    elem (numeros e) (elecciones ns) && valor e == [n]
```

11.1.2. Búsqueda de la solución por fuerza bruta

Divisiones de una lista

- (`divisiones xs`) es la lista de las divisiones de `xs` en dos listas no vacías. Por ejemplo,

```
*Main> divisiones "bcd"
[("b","cd"),("bc","d")]
*Main> divisiones "abcd"
[("a","bcd"),("ab","cd"),("abc","d")]
```

```
divisiones :: [a] -> [([a],[a])]
divisiones []     = []
divisiones [_]    = []
divisiones (x:xs) =
    ([x],xs) : [(x:is,ds) | (is,ds) <- divisiones xs]
```

Expresiones construibles

- (expresiones ns) es la lista de todas las expresiones construibles a partir de la lista de números ns. Por ejemplo,

```
*Main> expresiones [2,3,5]
[2+(3+5),2-(3+5),2*(3+5),2/(3+5),2+(3-5),2-(3-5),
 2*(3-5),2/(3-5),2+(3*5),2-(3*5),2*(3*5),2/(3*5),
 2+(3/5),2-(3/5),2*(3/5),2/(3/5),(2+3)+5,(2+3)-5,
 ...
```

```
expresiones :: [Int] -> [Expr]
expresiones []  = []
expresiones [n] = [Num n]
expresiones ns  = [e | (is,ds) <- divisiones ns
                     , i       <- expresiones is
                     , d       <- expresiones ds
                     , e       <- combina i d]
```

Combinación de expresiones

- (combina e1 e2) es la lista de las expresiones obtenidas combinando las expresiones e1 y e2 con una operación. Por ejemplo,

```
*Main> combina (Num 2) (Num 3)
[2+3,2-3,2*3,2/3]
```

```
combina :: Expr -> Expr -> [Expr]
combina e1 e2 = [Apl o e1 e2 | o <- ops]
```

Búsqueda de las soluciones

- (`soluciones ns n`) es la lista de las soluciones para la sucesión `ns` y objetivo `n` calculadas por fuerza bruta. Por ejemplo,

```
*Main> soluciones [1,3,7,10,25,50] 765
[3*((7*(50-10))-25), ((7*(50-10))-25)*3, ...
*Main> length (soluciones [1,3,7,10,25,50] 765)
780
*Main> length (soluciones [1,3,7,10,25,50] 831)
0
```

```
soluciones :: [Int] -> Int -> [Expr]
soluciones ns n =  [e | ns' <- elecciones ns
                      , e   <- expresiones ns'
                      , valor e == [n]]
```

Estadísticas de la búsqueda por fuerza bruta

- Estadísticas:

```
*Main> :set +s
*Main> head (soluciones [1,3,7,10,25,50] 765)
3*((7*(50-10))-25)
(8.47 secs, 400306836 bytes)
*Main> length (soluciones [1,3,7,10,25,50] 765)
780
(997.76 secs, 47074239120 bytes)
*Main> length (soluciones [1,3,7,10,25,50] 831)
0
(1019.13 secs, 47074535420 bytes)
*Main> :unset +s
```

11.1.3. Búsqueda combinando generación y evaluación

Resultados

- `Resultado` es el tipo de los pares formados por expresiones válidas y su valor.

```
type Resultado = (Expr,Int)
```

- (resultados ns) es la lista de todos los resultados construibles a partir de la lista de números ns. Por ejemplo,

```
*Main> resultados [2,3,5]
[(2+(3+5),10), (2*(3+5),16), (2+(3*5),17), (2*(3*5),30), ((2+3)+5,10),
 ((2+3)*5,25), ((2+3)/5,1),  ((2*3)+5,11), ((2*3)-5,1),  ((2*3)*5,30)]
```

```
resultados :: [Int] -> [Resultado]
resultados []  = []
resultados [n] = [(Num n,n) | n > 0]
resultados ns  = [res | (is,ds) <- divisiones ns
                      , ix      <- resultados is
                      , dy      <- resultados ds
                      , res     <- combina' ix dy]
```

Combinación de resultados

- (combina' r1 r2) es la lista de los resultados obtenidos combinando los resultados r1 y r2 con una operación. Por ejemplo,

```
*Main> combina' (Num 2,2) (Num 3,3)
[(2+3,5),(2*3,6)]
*Main> combina' (Num 3,3) (Num 2,2)
[(3+2,5),(3-2,1),(3*2,6)]
*Main> combina' (Num 2,2) (Num 6,6)
[(2+6,8),(2*6,12)]
*Main> combina' (Num 6,6) (Num 2,2)
[(6+2,8),(6-2,4),(6*2,12),(6/2,3)]
```

```
combina' :: Resultado -> Resultado -> [Resultado]
combina' (i,x) (d,y) =
    [(Apl o i d, aplica o x y) | o <- ops
                               , valida o x y]
```

Búsqueda combinando generación y evaluación

- (soluciones' ns n) es la lista de las soluciones para la sucesión ns y objetivo n calculadas intercalando generación y evaluación. Por ejemplo,

```
*Main> head (soluciones' [1,3,7,10,25,50] 765)
3*((7*(50-10))-25)
*Main> length (soluciones' [1,3,7,10,25,50] 765)
780
*Main> length (soluciones' [1,3,7,10,25,50] 831)
0
```

```
soluciones' :: [Int] -> Int -> [Expr]
soluciones' ns n = [e | ns'   <- elecciones ns
                      , (e,m) <- resultados ns'
                      , m == n]
```

Estadísticas de la búsqueda combinada

- Estadísticas:

```
*Main> head (soluciones' [1,3,7,10,25,50] 765)
3*((7*(50-10))-25)
(0.81 secs, 38804220 bytes)
*Main> length (soluciones' [1,3,7,10,25,50] 765)
780
(60.73 secs, 2932314020 bytes)
*Main> length (soluciones' [1,3,7,10,25,50] 831)
0
(61.68 secs, 2932303088 bytes)
```

11.1.4. Búsqueda mejorada mediante propiedades algebraicas

Aplicaciones válidas

- (`valida' o x y`) se verifica si la operación `o` aplicada a los números naturales `x` e `y` da un número natural, teniendo en cuenta las siguientes reducciones algebraicas

```
   x + y = y + x
   x * y = y * x
   x * 1 = x
   1 * y = y
   x / 1 = x
```

```
valida' :: Op -> Int -> Int -> Bool
valida' Sum x y = x <= y
```

```
valida' Res x y = x > y
valida' Mul x y = x /= 1 && y /= 1 && x <= y
valida' Div x y = y /= 0 && y /= 1 && x `mod` y == 0
```

Resultados válidos construibles

- `(resultados' ns)` es la lista de todos los resultados válidos construibles a partir de la lista de números `ns`. Por ejemplo,

```
*Main> resultados' [5,3,2]
[(5-(3-2),4),((5-3)+2,4),((5-3)*2,4),((5-3)/2,1)]
```

```
resultados' :: [Int] -> [Resultado]
resultados' []  = []
resultados' [n] = [(Num n,n) | n > 0]
resultados' ns  = [res | (is,ds) <- divisiones ns
                       , ix      <- resultados' is
                       , dy      <- resultados' ds
                       , res     <- combina'' ix dy]
```

Combinación de resultados válidos

- `(combina'' r1 r2)` es la lista de los resultados válidos obtenidos combinando los resultados `r1` y `r2` con una operación. Por ejemplo,

```
combina'' (Num 2,2) (Num 3,3)  =>  [(2+3,5),(2*3,6)]
combina'' (Num 3,3) (Num 2,2)  =>  [(3-2,1)]
combina'' (Num 2,2) (Num 6,6)  =>  [(2+6,8),(2*6,12)]
combina'' (Num 6,6) (Num 2,2)  =>  [(6-2,4),(6/2,3)]
```

```
combina'' :: Resultado -> Resultado -> [Resultado]
combina'' (i,x) (d,y) =
    [(Apl o i d, aplica o x y) | o <- ops
                               , valida' o x y]
```

Búsqueda mejorada mediante propiedades algebraicas

- `(soluciones'' ns n)` es la lista de las soluciones para la sucesión `ns` y objetivo `n` calculadas intercalando generación y evaluación y usando las mejoras aritméticas. Por ejemplo,

```
*Main> head (soluciones'' [1,3,7,10,25,50] 765)
3*((7*(50-10))-25)
*Main> length (soluciones'' [1,3,7,10,25,50] 765)
49
*Main> length (soluciones'' [1,3,7,10,25,50] 831)
0
```

```
soluciones'' :: [Int] -> Int -> [Expr]
soluciones'' ns n = [e | ns'   <- elecciones ns
                       , (e,m) <- resultados' ns'
                       , m == n]
```

Estadísticas de la búsqueda mejorada

- Estadísticas:

```
*Main> head (soluciones'' [1,3,7,10,25,50] 765)
3*((7*(50-10))-25)
(0.40 secs, 16435156 bytes)
*Main> length (soluciones'' [1,3,7,10,25,50] 765)
49
(10.30 secs, 460253716 bytes)
*Main> length (soluciones'' [1,3,7,10,25,50] 831)
0
(10.26 secs, 460253908 bytes)§
```

Comparación de las búsquedas

Comparación de las búsquedad problema de dados [1,3,7,10,25,50] obtener 765.

- Búsqueda de la primera solución:

```
               +---------------------+
               | segs. | bytes       |
+--------------+-------+-------------+
| soluciones   | 8.47  | 400.306.836 |
| soluciones'  | 0.81  |  38.804.220 |
| soluciones'' | 0.40  |  16.435.156 |
+--------------+-------+-------------+
```

Comparación de las búsquedas

- Búsqueda de todas las soluciones:

```
                +--------+---------------+
                | segs.  | bytes         |
+---------------+--------+---------------+
| soluciones    | 997.76 | 47.074.239.120 |
| soluciones'   |  60.73 |  2.932.314.020 |
| soluciones''  |  10.30 |    460.253.716 |
+---------------+--------+---------------+
```

Comparación de las búsquedas

Comprobación de que dados [1,3,7,10,25,50] no puede obtenerse 831

```
                +---------+---------------+
                | segs.   | bytes         |
+---------------+---------+---------------+
| soluciones    | 1019.13 | 47.074.535.420 |
| soluciones'   |   61.68 |  2.932.303.088 |
| soluciones''  |   10.26 |    460.253.908 |
+---------------+---------+---------------+
```

11.2. El problema de las reinas

El problema de las N reinas

- Enunciado: Colocar N reinas en un tablero rectangular de dimensiones N por N de forma que no se encuentren más de una en la misma línea: horizontal, vertical o diagonal.
- El problema se representa en el módulo `Reinas`. Importa la diferencia de conjuntos `(\\)` del módulo `List`:

```
module Reinas where
import Data.List ((\\))
```

- El tablero se representa por una lista de números que indican las filas donde se han colocado las reinas. Por ejemplo, `[3,5]` indica que se han colocado las reinas `(1,3)` y `(2,5)`.

```
type Tablero = [Int]
```

- `reinas n` es la lista de soluciones del problema de las N reinas. Por ejemplo, `reinas 4` ⇝ `[[3,1,4,2],[2,4,1,3]]`. La primera solución `[3,1,4,2]` se interpreta como

	R		
			R
R			
		R	

```
reinas :: Int -> [Tablero]
reinas n = aux n
    where aux 0     = [[]]
          aux (m+1) = [r:rs | rs <- aux m,
                              r <- ([1..n] \\ rs),
                              noAtaca r rs 1]
```

- `noAtaca r rs d` se verifica si la reina `r` no ataca a niguna de las de la lista `rs` donde la primera de la lista está a una distancia horizontal `d`.

```
noAtaca :: Int -> Tablero -> Int -> Bool
noAtaca _ [] _ = True
noAtaca r (a:rs) distH = abs(r-a) /= distH &&
                         noAtaca r rs (distH+1)
```

11.3. Números de Hamming

Números de Hamming

- Enunciado: Los números de Hamming forman una sucesión estrictamente creciente de números que cumplen las siguientes condiciones:
  1. El número 1 está en la sucesión.
  2. Si x está en la sucesión, entonces $2x$, $3x$ y $5x$ también están.
  3. Ningún otro número está en la sucesión.
- `hamming` es la sucesión de Hamming. Por ejemplo,

```
take 12 hamming ⇝ [1,2,3,4,5,6,8,9,10,12,15,16]
```

```
hamming :: [Int]
hamming = 1 : mezcla3 [2*i | i <- hamming]
                      [3*i | i <- hamming]
                      [5*i | i <- hamming]
```

- `mezcla3 xs ys zs` es la lista obtenida mezclando las listas ordenadas `xs`, `ys` y `zs` y eliminando los elementos duplicados. Por ejemplo,

```
Main> mezcla3 [2,4,6,8,10] [3,6,9,12] [5,10]
[2,3,4,5,6,8,9,10,12]
```

```
mezcla3 :: [Int] -> [Int] -> [Int] -> [Int]
mezcla3 xs ys zs = mezcla2 xs (mezcla2 ys zs)
```

- `mezcla2 xs ys zs` es la lista obtenida mezclando las listas ordenadas `xs` e `ys` y eliminando los elementos duplicados. Por ejemplo,

```
Main> mezcla2 [2,4,6,8,10,12] [3,6,9,12]
[2,3,4,6,8,9,10,12]
```

```
mezcla2 :: [Int] -> [Int] -> [Int]
mezcla2 p@(x:xs) q@(y:ys) | x < y     = x:mezcla2 xs q
                          | x > y     = y:mezcla2 p  ys
                          | otherwise = x:mezcla2 xs ys
mezcla2 []       ys                   = ys
mezcla2 xs       []                   = xs
```

Bibliografía

1. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
   - Cap. 11: The countdown problem.
2. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
   - Cap. 13: Puzzles y solitarios.

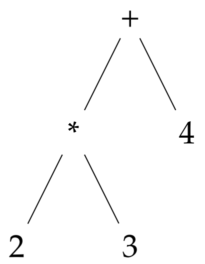
Tema 12

Analizadores sintácticos funcionales

12.1. Analizadores sintácticos

Analizadores sintácticos

- Un **analizador sintáctico** es un programa que analiza textos para determinar su **estructura sintáctica**.
- Ejemplo de análisis sintáctico aritmético: La estructura sintáctica de la cadena "2*3+4" es el árbol



- El análisis sintáctico forma parte del preprocesamiento en la mayoría de las aplicaciones reales.

12.2. El tipo de los analizadores sintácticos

Opciones para el tipo de los analizadores sintácticos

- Opción inicial:

```
type Analizador = String -> Tree
```

- Con la parte no analizada:

```
type Analizador = String -> (Tree,String)
```

- Con todos los análisis:

```
type Analizador = String -> [(Tree,String)]
```

- Con estructuras arbitrarias:

```
type Analizador a = String -> [(a,String)]
```

- Simplificación: analizadores que fallan o sólo dan un análisis.

12.3. Analizadores sintácticos básicos

Analizadores sintácticos básicos: resultado

- `(analiza a cs)` analiza la cadena `cs` mediante el analizador `a`. Por ejemplo,

```
analiza :: Analizador a -> String -> [(a,String)]
analiza a cs = a cs
```

- El analizador `resultado v` siempre tiene éxito, devuelve `v` y no consume nada. Por ejemplo,

```
*Main> analiza (resultado 1) "abc"
[(1,"abc")]
```

```
resultado :: a -> Analizador a
resultado v =  \xs -> [(v,xs)]
```

Analizadores sintácticos básicos: fallo

- El analizador `fallo` siempre falla. Por ejemplo,

```
*Main> analiza fallo "abc"
[]
```

```
fallo :: Analizador a
fallo =  \xs -> []
```

Analizadores sintácticos básicos: elemento

- El analizador `elemento` falla si la cadena es vacía y consume el primer elemento en caso contrario. Por ejemplo,

```
*Main> analiza elemento ""
[]
*Main> analiza elemento "abc"
[('a',"bc")]
```

```
elemento :: Analizador Char
elemento =  \xs -> case xs of
                     [] -> []
                     (x:xs) -> [(x , xs)]
```

12.4. Composición de analizadores sintácticos

12.4.1. Secuenciación de analizadores sintácticos

- `((p `liga` f) e)` falla si el análisis de `e` por `p` falla, en caso contrario, se obtiene un valor (`v`) y una salida (`s`), se aplica la función `f` al valor `v` obteniéndose un nuevo analizador con el que se analiza la salida `s`.

```
liga :: Analizador a ->
        (a -> Analizador b) ->
        Analizador b
p `liga` f = \ent -> case analiza p ent of
                       []         -> []
                       [(v,sal)] -> analiza (f v) sal
```

- `primeroTercero` es un analizador que devuelve los caracteres primero y tercero de la cadena. Por ejemplo,

```
primeroTercero "abel"  ⇝  [(('a','e'),"l")]
primeroTercero "ab"    ⇝  []
```

```
primeroTercero :: Analizador (Char,Char)
primeroTercero =
    elemento `liga` \x ->
    elemento `liga` \_ ->
    elemento `liga` \y ->
    resultado (x,y)
```

12.4.2. Elección de analizadores sintácticos

- `((p +++ q) e)` analiza `e` con `p` y si falla analiza `e` con `q`. Por ejemplo,

```
Main*> analiza (elemento +++ resultado 'd') "abc"
[('a',"bc")]
Main*> analiza (fallo +++ resultado 'd') "abc"
[('d',"abc")]
Main*> analiza (fallo +++ fallo) "abc"
[]
```

```
(+++) :: Analizador a -> Analizador a -> Analizador a
p +++ q = \ent -> case analiza p ent of
                    []        -> analiza q ent
                    [(v,sal)] -> [(v,sal)]
```

12.5. Primitivas derivadas

- `(sat p)` es el analizador que consume un elemento si dicho elemento cumple la propiedad `p` y falla en caso contrario. Por ejemplo,

```
analiza (sat isLower) "hola"  ⇝  [('h',"ola")]
analiza (sat isLower) "Hola"  ⇝  []
```

```
sat :: (Char -> Bool) -> Analizador Char
sat p = elemento `liga` \x ->
        if p x then resultado x else fallo
```

- `digito` analiza si el primer carácter es un dígito. Por ejemplo,

```
analiza digito "123"  ⇝  [('1',"23")]
analiza digito "uno"  ⇝  []
```

```
digito :: Analizador Char
digito = sat isDigit
```

- `minuscula` analiza si el primer carácter es una letra minúscula. Por ejemplo,

```
analiza minuscula "eva"  ⇝  [('e',"va")]
analiza minuscula "Eva"  ⇝  []
```

```
minuscula :: Analizador Char
minuscula = sat isLower
```

- `mayuscula` analiza si el primer carácter es una letra mayúscula. Por ejemplo,

```
analiza mayuscula "Eva"  ⇝  [('E',"va")]
analiza mayuscula "eva"  ⇝  []
```

```
mayuscula :: Analizador Char
mayuscula = sat isUpper
```

- `letra` analiza si el primer carácter es una letra. Por ejemplo,

```
analiza letra "Eva"  ⇝  [('E',"va")]
analiza letra "eva"  ⇝  [('e',"va")]
analiza letra "123"  ⇝  []
```

```
letra :: Analizador Char
letra = sat isAlpha
```

- `alfanumerico` analiza si el primer carácter es una letra o un número. Por ejemplo,

```
analiza alfanumerico "Eva"   ⇝  [('E',"va")]
analiza alfanumerico "eva"   ⇝  [('e',"va")]
analiza alfanumerico "123"   ⇝  [('1',"23")]
analiza alfanumerico " 123"  ⇝  []
```

```
alfanumerico :: Analizador Char
alfanumerico = sat isAlphaNum
```

- `(caracter x)` analiza si el primer carácter es igual al carácter `x`. Por ejemplo,

```
analiza (caracter 'E') "Eva"  ⇝  [('E',"va")]
analiza (caracter 'E') "eva"  ⇝  []
```

```
caracter :: Char -> Analizador Char
caracter x = sat (== x)
```

- `(cadena c)` analiza si empieza con la cadena `c`. Por ejemplo,

```
analiza (cadena "abc") "abcdef"  ⇝  [("abc","def")]
analiza (cadena "abc") "abdcef"  ⇝  []
```

```
cadena :: String -> Analizador String
cadena []     = resultado []
cadena (x:xs) = caracter x ‘liga‘ \x  ->
                cadena xs  ‘liga‘ \xs ->
                resultado (x:xs)
```

- `varios p` aplica el analizador p cero o más veces. Por ejemplo,

```
analiza (varios digito) "235abc"  ⇝  [("235","abc")]
analiza (varios digito) "abc235"  ⇝  [("","abc235")]
```

```
varios :: Analizador a -> Analizador [a]
varios p  = varios1 p +++ resultado []
```

- `varios1 p` aplica el analizador p una o más veces. Por ejemplo,

```
analiza (varios1 digito) "235abc"  ⇝  [("235","abc")]
analiza (varios1 digito) "abc235"  ⇝  []
```

```
varios1 :: Analizador a -> Analizador [a]
varios1 p = p        ‘liga‘ \v  ->
            varios p ‘liga‘ \vs ->
            resultado (v:vs)
```

- `ident` analiza si comienza con un identificador (i.e. una cadena que comienza con una letra minúscula seguida por caracteres alfanuméricos). Por ejemplo,

```
Main*> analiza ident "lunes12 de Ene"
[("lunes12"," de Ene")]
Main*> analiza ident "Lunes12 de Ene"
[]
```

```
ident :: Analizador String
ident =  minuscula           ‘liga‘ \x  ->
         varios alfanumerico ‘liga‘ \xs ->
         resultado (x:xs)
```

- `nat` analiza si comienza con un número natural. Por ejemplo,

```
analiza nat "14DeAbril"   ⇝  [(14,"DeAbril")]
analiza nat " 14DeAbril"  ⇝  []
```

```
nat :: Analizador Int
nat = varios1 digito `liga` \xs ->
      resultado (read xs)
```

- `espacio` analiza si comienza con espacios en blanco. Por ejemplo,

```
analiza espacio "    a b c"  ⇝  [((),"a b c")]
```

```
espacio :: Analizador ()
espacio = varios (sat isSpace) `liga` \_ ->
          resultado ()
```

12.6. Tratamiento de los espacios

- `unidad p` ignora los espacios en blanco y aplica el analizador `p`. Por ejemplo,

```
Main*> analiza (unidad nat) " 14DeAbril"
[(14,"DeAbril")]
Main*> analiza (unidad nat) " 14   DeAbril"
[(14,"DeAbril")]
```

```
unidad :: Analizador a -> Analizador a
unidad p = espacio `liga` \_ ->
           p       `liga` \v ->
           espacio `liga` \_ ->
           resultado v
```

- `identificador` analiza un identificador ignorando los espacios delante y detrás. Por ejemplo,

```
Main*> analiza identificador "  lunes12  de Ene"
[("lunes12","de Ene")]
```

```
identificador :: Analizador String
identificador = unidad ident
```

- `natural` analiza un número natural ignorando los espacios delante y detrás. Por ejemplo,

```
analiza natural "  14DeAbril"  ⇝  [(14,"DeAbril")]
```

```
natural :: Analizador Int
natural =  unidad nat
```

- `(simbolo xs)` analiza la cadena `xs` ignorando los espacios delante y detrás. Por ejemplo,

```
Main*> analiza (simbolo "abc") "  abcdef"
[("abc","def")]
```

```
simbolo :: String -> Analizador String
simbolo xs =  unidad (cadena xs)
```

- `listaNat` analiza una lista de naturales ignorando los espacios. Por ejemplo,

```
Main*> analiza listaNat " [  2,  3, 5   ]"
[([2,3,5],"")]
Main*> analiza listaNat " [  2,  3,]"
[]
```

```
listaNat :: Analizador [Int]
listaNat = simbolo "["          `liga` \_ ->
           natural              `liga` \n ->
           varios (simbolo ","  `liga` \_ ->
                   natural)     `liga` \ns ->
           simbolo "]"          `liga` \_ ->
           resultado (n:ns)
```

12.7. Analizador de expresiones aritméticas

Expresiones aritméticas

- Consideramos expresiones aritméticas:
  - construidas con números, operaciones ($+$ y $*$) y paréntesis.
  - $+$ y $*$ asocian por la derecha.

- $*$ tiene más prioridad que $+$.
- Ejemplos:
  - $2+3+5$ representa a $2+(3+5)$.
  - $2*3+5$ representa a $(2*3)+5$.

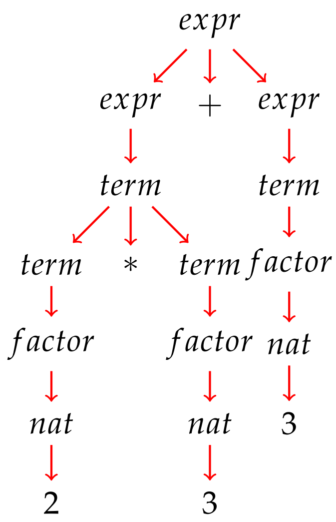
Gramáticas de las expresiones aritméticas: Gramática 1

- Gramática 1 de las expresiones aritméticas:
$$\begin{array}{lll} expr & ::= & expr + expr \mid expr * expr \mid (expr) \mid nat \\ nat & ::= & 0 \mid 1 \mid 2 \mid \dots \end{array}$$
- La gramática 1 no considera prioridad:
acepta $2+3*5$ como $(2+3)*5$ y como $2+(3*5)$
- La gramática 1 no considera asociatividad:
acepta $2+3+5$ como $(2+3)+5$ y como $2+(3+5)$
- La gramática 1 es ambigua.

Gramáticas de las expresiones aritméticas: Gramática 2

- Gramática 2 de las expresiones aritméticas (con prioridad):
$$\begin{array}{lll} expr & ::= & expr + expr \mid term \\ term & ::= & term * term \mid factor \\ factor & ::= & (expr) \mid nat \\ nat & ::= & 0 \mid 1 \mid 2 \mid \dots \end{array}$$
- La gramática 2 sí considera prioridad:
acepta $2+3*5$ sólo como $2+(3*5)$
- La gramática 2 no considera asociatividad:
acepta $2+3+5$ como $(2+3)+5$ y como $2+(3+5)$
- La gramática 2 es ambigua.

Árbol de análisis sintáctico de $2*3+5$ con la gramática 2



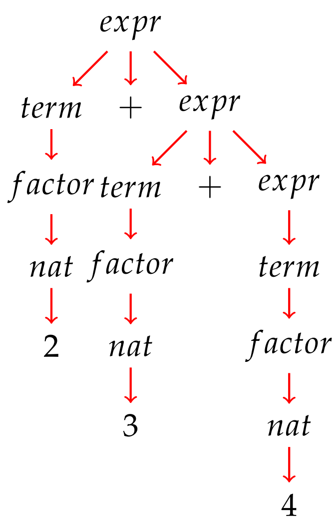
Gramáticas de las expresiones aritméticas: Gramática 3

- Gramática 3 de las expresiones aritméticas:

$$\begin{array}{lll} expr & ::= & term + expr \mid term \\ term & ::= & factor * term \mid factor \\ factor & ::= & (expr) \mid nat \\ nat & ::= & 0 \mid 1 \mid 2 \mid \ \dots \end{array}$$

- La gramática 3 sí considera prioridad:
  acepta $2 + 3 * 5$ sólo como $2 + (3 * 5)$
- La gramática 3 sí considera asociatividad:
  acepta $2 + 3 + 5$ como $2 + (3 + 5)$
- La gramática 3 no es ambigua (i.e. es libre de contexto).

Árbol de análisis sintáctico de $2 + 3 + 5$ con la gramática 3



Gramáticas de las expresiones aritméticas: Gramática 4

- La gramática 4 se obtiene simplificando la gramática 3:
  $$\begin{array}{lcl} expr & ::= & term\ (+\ expr \mid \epsilon) \\ term & ::= & factor\ (*\ term \mid \epsilon) \\ factor & ::= & (expr) \mid nat \\ nat & ::= & 0 \mid 1 \mid 2 \mid \ldots \end{array}$$
  donde ϵ es la cadena vacía.

- La gramática 4 no es ambigua.

- La gramática 4 es la que se usará para escribir el analizador de expresiones aritméticas.

Analizador de expresiones aritméticas

- expr analiza una expresión aritmética devolviendo su valor. Por ejemplo,

```
analiza expr "2*3+5"     ⇝  [(11,"")]
analiza expr "2*(3+5)"   ⇝  [(16,"")]
analiza expr "2+3*5"     ⇝  [(17,"")]
analiza expr "2*3+5abc"  ⇝  [(11,"abc")]
```

```
expr :: Analizador Int
expr = term              `liga` \t ->
       (simbolo "+"      `liga` \_ ->
        expr             `liga` \e ->
```

```
        resultado (t+e))
       +++ resultado t
```

- averbterm analiza un término de una expresión aritmética devolviendo su valor. Por ejemplo,

```
analiza term "2*3+5"      ⇝  [(6,"+5")]
analiza term "2+3*5"      ⇝  [(2,"+3*5")]
analiza term "(2+3)*5+7"  ⇝  [(25,"+7")]
```

```
term :: Analizador Int
term =  factor            `liga` \f ->
        (simbolo "*"      `liga` \_ ->
         term             `liga` \t ->
         resultado (f*t))
        +++ resultado f
```

- `factor` analiza un factor de una expresión aritmética devolviendo su valor. Por ejemplo,

```
analiza factor "2*3+5"      ⇝  [(2,"*3+5")]
analiza factor "(2+3)*5"    ⇝  [(5,"*5")]
analiza factor "(2+3*7)*5"  ⇝  [(23,"*5")]
```

```
factor :: Analizador Int
factor =  (simbolo "("  `liga` \_ ->
           expr         `liga` \e ->
           simbolo ")"  `liga` \_ ->
           resultado e)
          +++ natural
```

- `(valor cs)` analiza la cadena `cs` devolviendo su valor si es una expresión aritmética y un mensaje de error en caso contrario. Por ejemplo,

```
valor "2*3+5"      ⇝  11
valor "2*(3+5)"    ⇝  16
valor "2 * 3 + 5"  ⇝  11
valor "2*3x"       ⇝  *** Exception: sin usar x
valor "-1"         ⇝  *** Exception: entrada no valida
```

```
valor :: String -> Int
valor xs = case (analiza expr xs) of
             [(n,[])]  -> n
             [(_,sal)] -> error ("sin usar " ++ sal)
             []        -> error "entrada no valida"
```

Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
   - Cap. 11: Análisis sintáctico.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
   - Cap. 8: Functional parsers.
3. G. Hutton y E. Meijer. Monadic Parser Combinators. Technical Report NOTTCS–TR–96–4, Department of Computer Science, University of Nottingham, 1996.
4. G. Hutton y E. Meijer. Monadic Parsing in Haskell. Journal of Functional Programming, 8(4): 437—444, 1998.
5. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
   - Cap. 14: Analizadores.

Tema 13

Programas interactivos

13.1. Programas interactivos

- Los programas por lote no interactúan con los usuarios durante su ejecución.
- Los programas interactivos durante su ejecución pueden leer datos del teclado y escribir resultados en la pantalla.
- Problema:
  - Los programas interactivos tienen efectos laterales.
  - Los programa Haskell no tiene efectos laterales.

Ejemplo de programa interactivo

- Especificación: El programa pide una cadena y dice el número de caracteres que tiene.
- Ejemplo de sesión:

```
-- *Main> longitudCadena
-- Escribe una cadena: "Hoy es lunes"
-- La cadena tiene 14 caracteres
```

- Programa:

```
longitudCadena :: IO ()
longitudCadena = do putStr "Escribe una cadena: "
                    xs <- getLine
                    putStr "La cadena tiene "
                    putStr (show (length xs))
                    putStrLn " caracteres"
```

13.2. El tipo de las acciones de entrada/salida

- En Haskell se pueden escribir programas interactivos usando tipos que distingan las expresiones puras de las **acciones** impuras que tienen efectos laterales.
- `IO a` es el tipo de las acciones que devuelven un valor del tipo `a`.
- Ejemplos:
  - `IO Char` es el tipo de las acciones que devuelven un carácter.
  - `IO ()` es el tipo de las acciones que no devuelven ningún valor.

13.3. Acciones básicas

- `getChar :: IO Char`
  La acción `getChar` lee un carácter del teclado, lo muestra en la pantalla y lo devuelve como valor.
- `putChar :: c -> IO ()`
  La acción `putChar c` escribe el carácter `c` en la pantalla y no devuelve ningún valor.
- `return a -> IO a`
  La acción `return c` devuelve el valor `c` sin ninguna interacción.
- Ejemplo:

```
*Main> putChar 'b'
b*Main> it
()
```

13.4. Secuenciación

- Una sucesión de acciones puede combinarse en una acción compuesta mediante expresiones **do**.
- Ejemplo:

```
ejSecuenciacion :: IO (Char,Char)
ejSecuenciacion = do x <- getChar
                     getChar
                     y <- getChar
                     return (x,y)
```

Lee dos caracteres y devuelve el par formado por ellos. Por ejemplo,

```
*Main> ejSecuenciacion
b f
('b','f')
```

13.5. Primitivas derivadas

- Lectura de cadenas del teclado:

Prelude
```
getLine :: IO String
getLine = do x <- getChar
             if x == '\n' then return []
                else do xs <- getLine
                        return (x:xs)
```

- Escritura de cadenas en la pantalla:

Prelude
```
putStr :: String -> IO ()
putStr []     = return ()
putStr (x:xs) = do putChar x
                   putStr xs
```

- Escritura de cadenas en la pantalla y salto de línea:

Prelude
```
putStrLn :: String -> IO ()
putStrLn xs = do putStr xs
                 putChar '\n'
```

- Ejecución de una lista de acciones:

Prelude
```
sequence_ :: [IO a] -> IO ()
sequence_ []     = return ()
sequence_ (a:as) = do a
                      sequence_ as
```

Por ejemplo,

```
*Main> sequence_ [putStrLn "uno", putStrLn "dos"]
uno
dos
*Main> it
()
```

Ejemplo de programa con primitivas derivadas

- Especificación: El programa pide una cadena y dice el número de caracteres que tiene.
- Ejemplo de sesión:

```
-- *Main> longitudCadena
-- Escribe una cadena: "Hoy es lunes"
-- La cadena tiene 14 caracteres
```

- Programa:

```
longitudCadena :: IO ()
longitudCadena = do putStr "Escribe una cadena: "
                    xs <- getLine
                    putStr "La cadena tiene "
                    putStr (show (length xs))
                    putStrLn " caracteres"
```

13.6. Ejemplos de programas interactivos

13.6.1. Juego de adivinación interactivo

- Descripción: El programa le pide al jugador humano que piense un número entre 1 y 100 y trata de adivinar el número que ha pensado planteándole conjeturas a las que el jugador humano responde con mayor, menor o exacto según que el número pensado sea mayor, menor o igual que el número conjeturado por la máquina.
- Ejemplo de sesión:

```
Main> juego
Piensa un numero entre el 1 y el 100.
Es 50? [mayor/menor/exacto] mayor
Es 75? [mayor/menor/exacto] menor
Es 62? [mayor/menor/exacto] mayor
```

```
Es 68? [mayor/menor/exacto] exacto
Fin del juego
```

- Programa:

```
juego :: IO ()
juego =
    do putStrLn "Piensa un numero entre el 1 y el 100."
       adivina 1 100
       putStrLn "Fin del juego"

adivina :: Int -> Int -> IO ()
adivina a b =
    do putStr ("Es " ++ show conjetura ++ "? [mayor/menor/exacto] ")
       s <- getLine
       case s of
         "mayor"  -> adivina (conjetura+1) b
         "menor"  -> adivina a (conjetura-1)
         "exacto" -> return ()
         _        -> adivina a b
    where
      conjetura = (a+b) `div` 2
```

13.6.2. Calculadora aritmética

Acciones auxiliares

- Escritura de caracteres sin eco:

```
getCh :: IO Char
getCh = do hSetEcho stdin False
           c <- getChar
           hSetEcho stdin True
           return c
```

- Limpieza de la pantalla:

```
limpiaPantalla:: IO ()
limpiaPantalla= putStr "\ESC[2J"
```

- Escritura en una posición:

```
type Pos = (Int,Int)

irA :: Pos -> IO ()
irA (x,y) = putStr ("\ESC[" ++
                    show y ++ ";" ++ show x ++
                    "H")

escribeEn :: Pos -> String -> IO ()
escribeEn p xs = do irA p
                    putStr xs
```

Calculadora

```
calculadora :: IO ()
calculadora = do limpiaPantalla
                 escribeCalculadora
                 limpiar

escribeCalculadora :: IO ()
escribeCalculadora =
    do limpiaPantalla
       sequence_ [escribeEn (1,y) xs
                 | (y,xs) <- zip [1..13] imagenCalculadora]
       putStrLn ""
```

```
imagenCalculadora :: [String]
imagenCalculadora = ["+---------------+",
                     "|               |",
                     "+---+---+---+---+",
                     "| q | c | d | = |",
                     "+---+---+---+---+",
                     "| 1 | 2 | 3 | + |",
                     "+---+---+---+---+",
                     "| 4 | 5 | 6 | - |",
                     "+---+---+---+---+",
                     "| 7 | 8 | 9 | * |",
                     "+---+---+---+---+",
                     "| 0 | ( | ) | / |",
```

```
                          "+---+---+---+---+"]
```

Los primeros cuatro botones permiten escribir las órdenes:

- q para salir ('quit'),
- c para limpiar la agenda ('clear'),
- d para borrar un carácter ('delete') y
- = para evaluar una expresión.

Los restantes botones permiten escribir las expresiones.

```
limpiar :: IO ()
limpiar =  calc ""

calc :: String -> IO ()
calc xs = do escribeEnPantalla xs
             c <- getCh
             if elem c botones
                then procesa c xs
                else do calc xs

escribeEnPantalla xs =
    do escribeEn (3,2) "             "
       escribeEn (3,2) (reverse (take 13 (reverse xs)))
```

```
botones :: String
botones = standard ++ extra
    where
      standard = "qcd=123+456-789*0()/"
      extra    = "QCD \ESC\BS\DEL\n"

procesa :: Char -> String -> IO ()
procesa c xs
   | elem c "qQ\ESC"    = salir
   | elem c "dD\BS\DEL" = borrar xs
   | elem c "=\n"       = evaluar xs
   | elem c "cC"        = limpiar
   | otherwise          = agregar c xs
```

```
salir :: IO ()
salir = irA (1,14)

borrar :: String -> IO ()
borrar "" = calc ""
borrar xs = calc (init xs)

evaluar :: String -> IO ()
evaluar xs = case analiza expr xs of
             [(n,"")] -> calc (show n)
             _        -> do calc xs

agregar :: Char -> String -> IO ()
agregar c xs = calc (xs ++ [c])
```

13.6.3. El juego de la vida

Descripción del juego de la vida

- El tablero del juego de la vida es una malla formada por cuadrados ("células") que se pliega en todas las direcciones.
- Cada célula tiene 8 células vecinas, que son las que están próximas a ella, incluso en las diagonales.
- Las células tienen dos estados: están "vivas" o "muertas".
- El estado del tablero evoluciona a lo largo de unidades de tiempo discretas.
- Las transiciones dependen del número de células vecinas vivas:
  - Una célula muerta con exactamente 3 células vecinas vivas "nace" (al turno siguiente estará viva).
  - Una célula viva con 2 ó 3 células vecinas vivas sigue viva, en otro caso muere.

El tablero del juego de la vida

- Tablero:

```
type Tablero = [Pos]
```

- Dimensiones:

```
ancho :: Int
ancho = 5

alto :: Int
alto = 5
```

El juego de la vida

- Ejemplo de tablero:

```
ejTablero :: Tablero
ejTablero = [(2,3),(3,4),(4,2),(4,3),(4,4)]
```

- Representación del tablero:

```
 1234
1
2   O
3 O O
4  OO
```

- `(vida n t)` simula el juego de la vida a partir del tablero `t` con un tiempo entre generaciones proporcional a `n`. Por ejemplo,

```
vida 100000 ejTablero
```

```
vida :: Int -> Tablero -> IO ()
vida n t = do limpiaPantalla
              escribeTablero t
              espera n
              vida n (siguienteGeneracion t)
```

- Escritura del tablero:

```
escribeTablero :: Tablero -> IO ()
escribeTablero t = sequence_ [escribeEn p "O" | p <- t]
```

- Espera entre generaciones:

```
espera :: Int -> IO ()
espera n = sequence_ [return () | _ <- [1..n]]
```

- siguienteGeneracion t) es el tablero de la siguiente generación al tablero t. Por ejemplo,

```
*Main> siguienteGeneracion ejTablero
[(4,3),(3,4),(4,4),(3,2),(5,3)]
```

```
siguienteGeneracion :: Tablero -> Tablero
siguienteGeneracion t = supervivientes t ++ nacimientos t
```

- (supervivientes t) es la listas de posiciones de t que sobreviven; i.e. posiciones con 2 ó 3 vecinos vivos. Por ejemplo,

```
supervivientes ejTablero  ⇝  [(4,3),(3,4),(4,4)]
```

```
supervivientes :: Tablero -> [Pos]
supervivientes t = [p | p <- t,
                        elem (nVecinosVivos t p) [2,3]]
```

- (nVecinosVivos t c) es el número de vecinos vivos de la célula c en el tablero t. Por ejemplo,

```
nVecinosVivos ejTablero (3,3)  ⇝  5
nVecinosVivos ejTablero (3,4)  ⇝  3
```

```
nVecinosVivos :: Tablero -> Pos -> Int
nVecinosVivos t = length . filter (tieneVida t) . vecinos
```

(vecinos p) es la lista de los vecinos de la célula en la posición p. Por ejemplo,

```
vecinos (2,3) ⇝ [(1,2),(2,2),(3,2),(1,3),(3,3),(1,4),(2,4),(3,4)]
vecinos (1,2) ⇝ [(5,1),(1,1),(2,1),(5,2),(2,2),(5,3),(1,3),(2,3)]
vecinos (5,2) ⇝ [(4,1),(5,1),(1,1),(4,2),(1,2),(4,3),(5,3),(1,3)]
vecinos (2,1) ⇝ [(1,5),(2,5),(3,5),(1,1),(3,1),(1,2),(2,2),(3,2)]
vecinos (2,5) ⇝ [(1,4),(2,4),(3,4),(1,5),(3,5),(1,1),(2,1),(3,1)]
vecinos (1,1) ⇝ [(5,5),(1,5),(2,5),(5,1),(2,1),(5,2),(1,2),(2,2)]
vecinos (5,5) ⇝ [(4,4),(5,4),(1,4),(4,5),(1,5),(4,1),(5,1),(1,1)]
```

```
vecinos :: Pos -> [Pos]
vecinos (x,y) = map modular [(x-1,y-1), (x,y-1), (x+1,y-1),
                             (x-1,y),             (x+1,y),
                             (x-1,y+1), (x,y+1), (x+1,y+1)]
```

- (`modular p`) es la posición correspondiente a `p` en el tablero considerando los plegados. Por ejemplo,

```
modular (6,3)  ⇝  (1,3)
modular (0,3)  ⇝  (5,3)
modular (3,6)  ⇝  (3,1)
modular (3,0)  ⇝  (3,5)
```

```
modular :: Pos -> Pos
modular (x,y) = (((x-1) `mod` ancho) + 1, ((y-1) `mod` alto + 1))
```

- (`tieneVida t p`) se verifica si la posición `p` del tablero `t` tiene vida. Por ejemplo,

```
tieneVida ejTablero (1,1)  ⇝  False
tieneVida ejTablero (2,3)  ⇝  True
```

```
tieneVida :: Tablero -> Pos -> Bool
tieneVida t p = elem p t
```

- (`noTieneVida t p`) se verifica si la posición `p` del tablero `t` no tiene vida. Por ejemplo,

```
noTieneVida ejTablero (1,1)  ⇝  True
noTieneVida ejTablero (2,3)  ⇝  False
```

```
noTieneVida :: Tablero -> Pos -> Bool
noTieneVida t p = not (tieneVida t p)
```

- (`nacimientos t`) es la lista de los nacimientos de tablero `t`; i.e. las posiciones sin vida con 3 vecinos vivos. Por ejemplo,

```
nacimientos ejTablero  ⇝  [(3,2),(5,3)]
```

```
nacimientos' :: Tablero -> [Pos]
nacimientos' t = [(x,y) | x <- [1..ancho],
                          y <- [1..alto],
                          noTieneVida t (x,y),
                          nVecinosVivos t (x,y) == 3]
```

- Definición más eficiente de `nacimientos`

```
nacimientos :: Tablero -> [Pos]
nacimientos t = [p | p <- nub (concat (map vecinos t)),
                     noTieneVida t p,
                     nVecinosVivos t p == 3]
```

donde `(nub xs)` es la lista obtenida eliminando las repeticiones de `xs`. Por ejemplo,

```
nub [2,3,2,5]  ⇝  [2,3,5]
```

Bibliografía

1. H. Daumé III. Yet Another Haskell Tutorial. 2006.
   - Cap. 5: Basic Input/Output.
2. G. Hutton. *Programming in Haskell*. Cambridge University Press, 2007.
   - Cap. 9: Interactive programs.
3. B. O'Sullivan, J. Goerzen y D. Stewart. *Real World Haskell*. O'Reilly, 2009.
   - Cap. 7: I/O.
4. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
   - Cap. 7: Entrada y salida.
5. S. Thompson. *Haskell: The Craft of Functional Programming*, Second Edition. Addison-Wesley, 1999.
   - Cap. 18: Programming with actions.

Tema 14

El TAD de las pilas

14.1. Tipos abstractos de datos

14.1.1. Abstracción y tipos abstractos de datos

Abstracción y tipos abstractos de datos

- La **abstracción** es un mecanismo para comprender problemas que involucran una gran cantidad de detalles.
- Aspectos de la abstracción:
  - **Destacar** los detalles relevantes.
  - **Ocultar** los detalles irrelevantes.
- Un **tipo abstracto de datos** (TAD) es una colección de *valores* y *operaciones* que se definen mediante una *especificación* que es independiente de cualquier *representación*.
- Un TAD es una abstracción:
  - Se destacan los detalles (normalmente pocos) de la especificación (*el qué*).
  - Se ocultan los detalles (normalmente numerosos) de la implementación (*el cómo*).
- Analogía con las **estructuras algebraicas**.

14.2. Especificación del TAD de las pilas

14.2.1. Signatura del TAD pilas

Descripción informal de las pilas

- Una **pila** es una estructura de datos, caracterizada por ser una secuencia de elementos en la que las operaciones de inserción y extracción se realizan por el mismo extremo.
- La pilas también se llaman estructuras LIFO (del inglés Last In First Out), debido a que el último elemento en entrar será el primero en salir.
- Analogía con las pilas de platos.

Signatura del TAD de las pilas

- Signatura:

```
vacia    :: Pila a
apila    :: a -> Pila a -> Pila a
cima     :: Pila a -> a
desapila :: Pila a -> Pila a
esVacia  :: Pila a -> Bool
```

- Descripción:
  - `vacia` es la pila vacía.
  - `(apila x p)` es la pila obtenida añadiendo `x` al principio de `p`.
  - `(cima p)` es la cima de la pila `p`.
  - `(desapila p)` es la pila obtenida suprimiendo la cima de `p`.
  - `(esVacia p)` se verifica si `p` es la pila vacía.

14.2.2. Propiedades del TAD de las pilas

Propiedades de las pilas

1. `cima (apila x p) == x`
2. `desapila (apila x p) == p`
3. `esVacia vacia`
4. `not (esVacia (apila x p))`

14.3. Implementaciones del TAD de las pilas

14.3.1. Las pilas como tipos de datos algebraicos

- Cabecera del módulo:

```
module PilaConTipoDeDatoAlgebraico
    (Pila,
     vacia,    -- Pila a
     apila,    -- a -> Pila a -> Pila a
     cima,     -- Pila a -> a
     desapila, -- Pila a -> Pila a
     esVacia   -- Pila a -> Bool
    ) where
```

- Tipo de dato algebraico de las pilas.

```
data Pila a = Vacia | P a (Pila a)
              deriving Eq
```

- Procedimiento de escritura de pilas.

```
instance (Show a) => Show (Pila a) where
    showsPrec p Vacia cad   = showChar '-' cad
    showsPrec p (P x s) cad =
       shows x (showChar '|' (shows s cad))
```

- Ejemplo de pila:
  - Definición

    ```
    p1 :: Pila Int
    p1 = apila 1 (apila 2 (apila 3 vacia))
    ```

  - Sesión

    ```
    ghci> p1
    1|2|3|-
    ```

- `vacia` es la pila vacía. Por ejemplo,

```
ghci> vacia
-
```

```
vacia :: Pila a
vacia = Vacia
```

- (`apila x p`) es la pila obtenida añadiedo `x` encima de la pila `p`. Por ejemplo,

```
apila 4 p1  =>  4|1|2|3|-
```

```
apila :: a -> Pila a -> Pila a
apila x p = P x p
```

- (`cima p`) es la cima de la pila `p`. Por ejemplo,

```
cima p1  ==  1
```

```
cima :: Pila a -> a
cima Vacia   = error "cima: pila vacia"
cima (P x _) =  x
```

- (`desapila p`) es la pila obtenida suprimiendo la cima de la pila `p`. Por ejemplo,

```
desapila p1  =>  2|3|-
```

```
desapila :: Pila a -> Pila a
desapila Vacia   = error "desapila: pila vacia"
desapila (P _ p) = p
```

- (`esVacia p`) se verifica si `p` es la pila vacía. Por ejemplo,

```
esVacia p1     ==  False
esVacia vacia  ==  True
```

```
esVacia :: Pila a -> Bool
esVacia Vacia = True
esVacia _     = False
```

14.3.2. Las pilas como listas

- Cabecera del módulo

```
module PilaConListas
    (Pila,
     vacia,      -- Pila a
     apila,      -- a -> Pila a -> Pila a
     cima,       -- Pila a -> a
     desapila,   -- Pila a -> Pila a
     esVacia     -- Pila a -> Bool
    ) where
```

- Tipo de datos de las pilas:

```
newtype Pila a = P [a]
    deriving Eq
```

- Procedimiento de escritura de pilas.

```
instance (Show a) => Show (Pila a) where
    showsPrec p (P [])     cad = showChar '-' cad
    showsPrec p (P (x:xs)) cad
        = shows x (showChar '|' (shows (P xs) cad))
```

- Ejemplo de pila: p1 es la pila obtenida anadiéndole los elementos 3, 2 y 1 a la pila vacía. Por ejemplo,

```
ghci> p1
1|2|3|-
```

```
p1 = apila 1 (apila 2 (apila 3 vacia))
```

- vacia es la pila vacía. Por ejemplo,

```
ghci> vacia
-
```

```
vacia   :: Pila a
vacia = P []
```

- (apila x p) es la pila obtenida añadiendo x encima de la pila p. Por ejemplo,

```
apila 4 p1  =>  4|1|2|3|-|
```

```
apila :: a -> Pila a -> Pila a
apila x (P xs) = P (x:xs)
```

- (`cima p`) es la cima de la pila p. Por ejemplo,

```
cima p1  ==  1
```

```
cima :: Pila a -> a
cima (P (x:_)) = x
cima (P [])    = error "cima de la pila vacia"
```

- (`desapila p`) es la pila obtenida suprimiendo la cima de la pila p. Por ejemplo,

```
desapila p1  =>  2|3|-
```

```
desapila :: Pila a -> Pila a
desapila (P [])     = error "desapila la pila vacia"
desapila (P (_:xs)) = P  xs
```

- (`esVacia p`) se verifica si p es la pila vacía. Por ejemplo,

```
esVacia p1     ==  False
esVacia vacia  ==  True
```

```
esVacia :: Pila a -> Bool
esVacia (P xs) = null xs
```

14.4. Comprobación de las implementaciones con QuickCheck

14.4.1. Librerías auxiliares

Importación de librerías

- Importación de la implementación de pilas que se desea comprobar.

```
import PilaConTipoDeDatoAlgebraico
-- import PilaConListas
```

- Importación de las librerías de comprobación

```
import Test.QuickCheck
import Test.Framework
import Test.Framework.Providers.QuickCheck2
```

14.4.2. Generador de pilas

Generador de pilas

- genPila es un generador de pilas. Por ejemplo,

```
ghci> sample genPila
0|0|-
-6|4|-3|3|0|-
-
9|5|-1|-3|0|-8|-5|-7|2|-
...
```

```
genPila :: (Num a, Arbitrary a) => Gen (Pila a)
genPila = do xs <- listOf arbitrary
             return (foldr apila vacia xs)

instance (Arbitrary a, Num a) => Arbitrary (Pila a) where
    arbitrary = genPila
```

14.4.3. Especificación de las propiedades de las pilas

- La cima de la pila que resulta de añadir x a la pila p es x.

```
prop_cima_apila :: Int -> Pila Int -> Bool
prop_cima_apila x p =
    cima (apila x p) == x
```

- La pila que resulta de desapilar después de añadir cualquier elemento a una pila p es p.

```
prop_desapila_apila :: Int -> Pila Int -> Bool
prop_desapila_apila x p =
    desapila (apila x p) == p
```

- La pila vacía está vacía.

```
prop_vacia_esta_vacia :: Bool
prop_vacia_esta_vacia =
    esVacia vacia
```

- La pila que resulta de añadir un elemento en un pila cualquiera no es vacía.

```
prop_apila_no_es_vacia :: Int -> Pila Int -> Bool
prop_apila_no_es_vacia x p =
    not (esVacia (apila x p))
```

14.4.4. Comprobación de las propiedades

Definición del procedimiento de comprobación

- `compruebaPropiedades` comprueba todas las propiedades con la plataforma de verificación.

```
compruebaPropiedades =
    defaultMain
        [testGroup "Propiedades del TAD pilas"
          [testProperty "P1" prop_cima_apila,
           testProperty "P2" prop_desapila_apila,
           testProperty "P3" prop_vacia_esta_vacia,
           testProperty "P4" prop_apila_no_es_vacia]]
```

Comprobación de las propiedades de las pilas

```
ghci> compruebaPropiedades
Propiedades del TAD pilas:
  P1: [OK, passed 100 tests]
  P2: [OK, passed 100 tests]
  P3: [OK, passed 100 tests]
  P4: [OK, passed 100 tests]

         Properties  Total
 Passed  4           4
 Failed  0           0
 Total   4           4
```

Tema 15

El TAD de las colas

15.1. Especificación del TAD de las colas

15.1.1. Signatura del TAD de las colas

Descripción informal de las colas

- Una **cola** es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción se realiza por un extremo (el posterior o final) y la operación de extracción por el otro (el anterior o frente).
- Las colas también se llaman estructuras FIFO (del inglés First In First Out), debido a que el primer elemento en entrar será también el primero en salir.
- Analogía con las colas del cine.

Signatura del TAD colas

- Signatura:

```
vacia   :: Cola a
inserta :: a -> Cola a -> Cola a
primero :: Cola a -> a
resto   :: Cola a -> Cola a
esVacia :: Cola a -> Bool
valida  :: Cola a -> Bool
```

- Descripción de las operaciones:
  - `vacia` es la cola vacía.
  - `(inserta x c)` es la cola obtenida añadiendo `x` al final de `c`.

- (primero c) es el primero de la cola c.
- (resto c) es la cola obtenida eliminando el primero de c.
- (esVacia c) se verifica si c es la cola vacía.
- (valida c) se verifica si c representa una cola válida.

15.1.2. Propiedades del TAD de las colas

Propiedades del TAD de las colas

1. `primero (inserta x vacia) == x`
2. Si c es una cola no vacía, entonces
   `primero (inserta x c) == primero c,`
3. `resto (inserta x vacia) == vacia`
4. Si c es una cola no vacía, entonces
   `resto (inserta x c) == inserta x (resto c)`
5. `esVacia vacia`
6. `not (esVacia (inserta x c))`

15.2. Implementaciones del TAD de las colas

15.2.1. Implementación de las colas mediante listas

- Cabecera del módulo:

```
module ColaConListas
    (Cola,
     vacia,   -- Cola a
     inserta, -- a -> Cola a -> Cola a
     primero, -- Cola a -> a
     resto,   -- Cola a -> Cola a
     esVacia, -- Cola a -> Bool
     valida   -- Cola a -> Bool
    ) where
```

- Representación de las colas mediante listas:

```
newtype Cola a = C [a] deriving (Show, Eq)
```

- Ejemplo de cola: `c1` es la cola obtenida añadiéndole a la cola vacía los números del 1 al 10. Por ejemplo,

```
ghci> c1
C [10,9,8,7,6,5,4,3,2,1]
```

```
c1 = foldr inserta vacia [1..10]
```

- `vacia` es la cola vacía. Por ejemplo,

```
ghci> vacia
C []
```

```
vacia :: Cola a
vacia = C []
```

- `(inserta x c)` es la cola obtenida añadiendo `x` al final de la cola `c`. Por ejemplo,

```
inserta 12 c1  ⇝  C [10,9,8,7,6,5,4,3,2,1,12]
```

```
inserta :: a -> Cola a -> Cola a
inserta x (C c) = C (c ++ [x])
```

- `(primero c)` es el primer elemento de la cola `c`. Por ejemplo,

```
primero c1  ⇝  10
```

```
primero :: Cola a -> a
primero (C (x:_)) = x
primero (C [])    = error "primero: cola vacia"
```

- `(resto c)` es la cola obtenida eliminando el primer elemento de la cola `c`. Por ejemplo,

```
resto c1  ⇝  C [9,8,7,6,5,4,3,2,1]
```

```
resto :: Cola a -> Cola a
resto (C (_:xs)) = C xs
resto (C [])     = error "resto: cola vacia"
```

- `(esVacia c)` se verifica si `c` es la cola vacía. Por ejemplo,

```
esVacia c1     ⇝  False
esVacia vacia  ⇝  True
```

```
esVacia :: Cola a -> Bool
esVacia (C xs)  = null xs
```

- `(valida c)` se verifica si `c` representa una cola válida. Con esta representación, todas las colas son válidas.

```
valida :: Cola a -> Bool
valida c = True
```

15.2.2. Implementación de las colas mediante pares de listas

Las colas como pares de listas

- En esta implementación, una cola `c` se representa mediante un par de listas `(xs,ys)` de modo que los elementos de `c` son, en ese orden, los elementos de la lista `xs++(reverse ys)`.
- Al dividir la lista en dos parte e invertir la segunda de ellas, esperamos hacer más eficiente las operaciones sobre las colas.
- Impondremos también una restricción adicional sobre la representación: las colas serán representadas mediante pares `(xs,ys)` tales que si `xs` es vacía, entonces `ys` será también vacía.
- Esta restricción ha de mantenerse por las operaciones que crean colas.

Implementación de las colas como pares de listas

- Cabecera del módulo

```
module ColaConDosListas
    (Cola,
     vacia,   -- Cola a
     inserta, -- a -> Cola a -> Cola a
     primero, -- Cola a -> a
     resto,   -- Cola a -> Cola a
     esVacia, -- Cola a -> Bool
     valida   -- Cola a -> Bool
    ) where
```

- Las colas como pares de listas

```
newtype Cola a = C ([a],[a])
```

- (`valida c`) se verifica si la cola `c` es válida; es decir, si su primer elemento es vacío entonces también lo es el segundo. Por ejemplo,

```
valida (C ([2],[5]))  ⇝  True
valida (C ([2],[]))   ⇝  True
valida (C ([],[5]))   ⇝  False
```

```
valida:: Cola a -> Bool
valida (C (xs,ys)) = not (null xs) || null ys
```

- Procedimiento de escritura de colas como pares de listas.

```
instance Show a => Show (Cola a) where
    showsPrec p (C (xs,ys)) cad
        = showString "C " (showList (xs ++ (reverse ys)) cad)
```

- Ejemplo de cola: `c1` es la cola obtenida añadiéndole a la cola vacía los números del 1 al 10. Por ejemplo,

```
ghci> c1
C [10,9,8,7,6,5,4,3,2,1]
```

```
c1 :: Cola Int
c1 = foldr inserta vacia [1..10]
```

- `vacia` es la cola vacía. Por ejemplo,

```
ghci> c1
C [10,9,8,7,6,5,4,3,2,1]
```

```
vacia :: Cola a
vacia  = C ([],[])
```

- (`inserta x c`) es la cola obtenida añadiendo `x` al final de la cola `c`. Por ejemplo,

```
inserta 12 c1  ⇝  C [10,9,8,7,6,5,4,3,2,1,12]
```

```
inserta :: a -> Cola a -> Cola a
inserta y (C (xs,ys)) = C (normaliza (xs,y:ys))
```

- `(normaliza p)` es la cola obtenida al normalizar el par de listas `p`. Por ejemplo,

```
normaliza ([],[2,5,3])   ⇝  ([3,5,2],[])
normaliza ([4],[2,5,3])  ⇝  ([4],[2,5,3])
```

```
normaliza :: ([a],[a]) -> ([a],[a])
normaliza ([], ys) = (reverse ys, [])
normaliza p        = p
```

- `(primero c)` es el primer elemento de la cola `c`. Por ejemplo,

```
primero c1  ⇝  10
```

```
primero  :: Cola a -> a
primero (C (x:xs,ys)) = x
primero _             = error "primero: cola vacia"
```

- `(resto c)` es la cola obtenida eliminando el primer elemento de la cola `c`. Por ejemplo,

```
resto c1  ⇝  C [9,8,7,6,5,4,3,2,1]
```

```
resto  :: Cola a -> Cola a
resto (C (x:xs,ys)) = C (normaliza (xs,ys))
resto (C ([],[]))   = error "resto: cola vacia"
```

- `(esVacia c)` se verifica si `c` es la cola vacía. Por ejemplo,

```
esVacia c1     ⇝  False
esVacia vacia  ⇝  True
```

```
esVacia :: Cola a -> Bool
esVacia (C (xs,_)) = null xs
```

- `(elementos c)` es la lista de los elementos de la cola `c` en el orden de la cola. Por ejemplo,

```
elementos (C ([3,2],[5,4,7])) ⇝ [3,2,7,4,5]
```

```
elementos:: Cola a -> [a]
elementos (C (xs,ys)) = xs ++ (reverse ys)
```

- (igualColas c1 c2) se verifica si las colas c1 y c2 son iguales.

```
ghci> igualColas (C ([3,2],[5,4,7])) (C ([3],[5,4,7,2]))
True
ghci> igualColas (C ([3,2],[5,4,7])) (C ([],[5,4,7,2,3]))
False
```

```
igualColas c1 c2 =
    valida c1 && valida c2 &&
    elementos c1 == elementos c2
```

- Extensión de la igualdad a las colas:

```
instance (Eq a) => Eq (Cola a) where
    (==) = igualColas
```

15.3. Comprobación de las implementaciones con QuickCheck

15.3.1. Librerías auxiliares

Importación de librerías

- Importación de la implementación de las colas que se desea comprobar.

```
import ColaConListas
-- import ColaConDosListas
```

- Importación de librerías auxiliares

```
import Data.List
import Test.QuickCheck
import Test.Framework
import Test.Framework.Providers.QuickCheck2
```

15.3.2. Generador de colas

Generador de colas

- `genCola` es un generador de colas de enteros. Por ejemplo,

```
ghci> sample genCola
C ([7,8,4,3,7],[5,3,3])
C ([1],[13])
...
```

```
genCola :: Gen (Cola Int)
genCola = frequency [(1, return vacia),
                     (30, do n <- choose (10,100)
                             xs <- vectorOf n arbitrary
                             return (creaCola xs))]
          where creaCola = foldr inserta vacia

instance Arbitrary (Cola Int) where
    arbitrary = genCola
```

Corrección del generador de colas

- Propiedad: Todo los elementos generados por `genCola` son colas válidas.

```
prop_genCola_correcto :: Cola Int -> Bool
prop_genCola_correcto c = valida c
```

- Comprobación.

```
ghci> quickCheck prop_genCola_correcto
+++ OK, passed 100 tests.
```

15.3.3. Especificación de las propiedades de las colas

- El primero de la cola obtenida añadiendo `x` a la cola vacía es `x`.

```
prop_primero_inserta_vacia :: Int -> Bool
prop_primero_inserta_vacia x =
    primero (inserta x vacia) == x
```

- Si una cola no está vacía, su primer elemento no varía al añadirle un elemento.

```
prop_primero_inserta_no_vacia :: Cola Int -> Int -> Int
                                  -> Bool
prop_primero_inserta_no_vacia c x y =
    primero (inserta x c') == primero c'
    where c' = inserta y vacia
```

- El resto de la cola obtenida insertando un elemento en la cola vacía es la cola vacía.

```
prop_resto_inserta_vacia :: Int -> Bool
prop_resto_inserta_vacia x =
    resto (inserta x vacia) == vacia
```

- Las operaciones de encolar y desencolar conmutan.

```
prop_resto_inserta_en_no_vacia :: Cola Int -> Int -> Int
                                   -> Bool
prop_resto_inserta_en_no_vacia c x y =
    resto (inserta x c') == inserta x (resto c')
    where c' = inserta y c
```

- `vacia` es una cola vacía.

```
prop_vacia_es_vacia :: Bool
prop_vacia_es_vacia =
    esVacia vacia
```

- La cola obtenida insertando un elemento no es vacía.

```
prop_inserta_no_es_vacia :: Int -> Cola Int -> Bool
prop_inserta_no_es_vacia x c =
    not (esVacia (inserta x c))
```

- La cola vacía es válida.

```
prop_valida_vacia :: Bool
prop_valida_vacia = valida vacia
```

- Al añadirle un elemento a una cola válida se obtiene otra válida.

```
prop_valida_inserta :: Cola Int -> Int -> Property
prop_valida_inserta c x =
    valida c ==> valida (inserta x c)
```

- El resto de una cola válida y no vacía es una cola válida.

```
prop_valida_resto :: Cola Int -> Property
prop_valida_resto c =
    valida c && not (esVacia c) ==> valida (resto c)
```

15.3.4. Comprobación de las propiedades

Definición del procedimiento de comprobación

- `compruebaPropiedades` comprueba todas las propiedades con la plataforma de verificación.

```
compruebaPropiedades =
    defaultMain
        [testGroup "Propiedades del TAD cola"
        [testGroup "Propiedades del TAD cola"
          [testProperty "P1" prop_primero_inserta_vacia,
           testProperty "P2" prop_primero_inserta_no_vacia,
           testProperty "P3" prop_resto_inserta_vacia,
           testProperty "P4" prop_resto_inserta_en_no_vacia,
           testProperty "P5" prop_vacia_es_vacia,
           testProperty "P6" prop_inserta_no_es_vacia,
           testProperty "P7" prop_valida_vacia,
           testProperty "P8" prop_valida_inserta,
           testProperty "P9" prop_valida_resto]]
```

Comprobación de las propiedades de las colas

```
ghci> compruebaPropiedades
Propiedades del TAD cola
  P1: [OK, passed 100 tests]
  P2: [OK, passed 100 tests]
  P3: [OK, passed 100 tests]
  P4: [OK, passed 100 tests]
```

```
 P5: [OK, passed 100 tests]
 P6: [OK, passed 100 tests]
 P7: [OK, passed 100 tests]
 P8: [OK, passed 100 tests]
 P9: [OK, passed 100 tests]

        Properties  Total
Passed  9           9
Failed  0           0
Total   9           9
```

Tema 16

El TAD de las colas de prioridad

16.1. Especificación del TAD de las colas de prioridad

16.1.1. Signatura del TAD colas de prioridad

Descripción de las colas de prioridad

- Una **cola de prioridad** es una cola en la que cada elemento tiene asociada una prioridad. La operación de extracción siempre elige el elemento de menor prioridad.
- Ejemplos:
  - La cola de las ciudades ordenadas por su distancia al destino final.
  - Las colas de las tareas pendientes ordenadas por su fecha de terminación.

Signatura de las colas de prioridad

- Signatura:

```
vacia,   :: Ord a => CPrioridad a
inserta, :: Ord a => a -> CPrioridad a -> CPrioridad a
primero, :: Ord a => CPrioridad a -> a
resto,   :: Ord a => CPrioridad a -> CPrioridad a
esVacia, :: Ord a => CPrioridad a -> Bool
valida   :: Ord a => CPrioridad a -> Bool
```

- Descripción de las operaciones:
  - `vacia` es la cola de prioridad vacía.
  - `(inserta x c)` añade el elemento `x` a la cola de prioridad `c`.

- (`primero c`) es el primer elemento de la cola de prioridad `c`.
- (`resto c`) es el resto de la cola de prioridad `c`.
- (`esVacia c`) se verifica si la cola de prioridad `c` es vacía.
- (`valida c`) se verifica si `c` es una cola de prioridad válida.

16.1.2. Propiedades del TAD de las colas de prioridad

1. `inserta x (inserta y c) == inserta y (inserta x c)`
2. `primero (inserta x vacia) == x`
3. Si `x <= y`, entonces
   `primero (inserta y (inserta x c))`
   `== primero (inserta x c)`
4. `resto (inserta x vacia) == vacia`
5. Si `x <= y`, entonces
   `resto (inserta y (inserta x c))`
   `== inserta y (resto (inserta x c))`
6. `esVacia vacia`
7. `not (esVacia (inserta x c))`

16.2. Implementaciones del TAD de las colas de prioridad

16.2.1. Las colas de prioridad como listas

- Cabecera del módulo:

```
module ColaDePrioridadConListas
    (CPrioridad,
     vacia,   -- Ord a => CPrioridad a
     inserta, -- Ord a => a -> CPrioridad a -> CPrioridad a
     primero, -- Ord a => CPrioridad a -> a
     resto,   -- Ord a => CPrioridad a -> CPrioridad a
     esVacia, -- Ord a => CPrioridad a -> Bool
     valida   -- Ord a => CPrioridad a -> Bool
    ) where
```

- Colas de prioridad mediante listas:

```
newtype CPrioridad a = CP [a]
    deriving (Eq, Show)
```

- Ejemplo de cola de prioridad: `cp1` es la cola de prioridad obtenida añadiéndole a la cola vacía los elementos 3, 1, 7, 2 y 9.

```
cp1  ⇝ CP [1,2,3,7,9]
```

```
cp1 :: CPrioridad Int
cp1 = foldr inserta vacia [3,1,7,2,9]
```

- `(valida c)` se verifica si `c` es una cola de prioridad válida; es decir, está ordenada crecientemente. Por ejemplo,

```
valida (CP [1,3,5])  ⇝  True
valida (CP [1,5,3])  ⇝  False
```

```
valida :: Ord a => CPrioridad a -> Bool
valida (CP xs) = ordenada xs
    where ordenada (x:y:zs) = x <= y && ordenada (y:zs)
          ordenada _        = True
```

- `vacia` es la cola de prioridad vacía. Por ejemplo,

```
vacia  ⇝  CP []
```

```
vacia :: Ord a => CPrioridad a
vacia = CP []
```

- `(inserta x c)` es la cola obtenida añadiendo el elemento `x` a la cola de prioridad `c`. Por ejemplo,

```
cp1            ⇝  CP [1,2,3,7,9]
inserta 5 cp1  ⇝  CP [1,2,3,5,7,9]
```

```
inserta :: Ord a => a -> CPrioridad a -> CPrioridad a
inserta x (CP q) = CP (ins x q)
    where ins x []                   = [x]
          ins x r@(e:r') | x < e     = x:r
                         | otherwise = e:ins x r'
```

- (primero c) es el primer elemento de la cola de prioridad c.

```
cp1          ⇝  CP [1,2,3,7,9]
primero cp1  ⇝  1
```

```
primero :: Ord a => CPrioridad a -> a
primero (CP(x:_)) = x
primero _         = error "primero: cola vacia"
```

- (resto c) es la cola de prioridad obtenida eliminando el primer elemento de la cola de prioridad c. Por ejemplo,

```
cp1        ⇝  CP [1,2,3,7,9]
resto cp1  ⇝  CP [2,3,7,9]
```

```
resto :: Ord a => CPrioridad a -> CPrioridad a
resto (CP (_:xs)) = CP xs
resto _           = error "resto: cola vacia"
```

- (esVacia c) se verifica si la cola de prioridad c es vacía. Por ejemplo,

```
esVacia cp1    ⇝  False
esVacia vacia  ⇝  True
```

```
esVacia :: Ord a => CPrioridad a -> Bool
esVacia (CP xs) = null xs
```

16.2.2. Las colas de prioridad como montículos

La implementación de las colas de prioridad como montículos (ColaDePrioridadConMonticulos) se encuentra en en el tema 20 (El TAD de los montículos).

16.3. Comprobación de las implementaciones con QuickCheck

16.3.1. Librerías auxiliares

- Importación de la implementación de colas de prioridad que se desea verificar.

```
import ColaDePrioridadConListas
-- ColaDePrioridadConMonticulos.hs
```

- Importación de las librerías de comprobación

```
import Test.QuickCheck
import Test.Framework
import Test.Framework.Providers.QuickCheck2
```

16.3.2. Generador de colas de prioridad

- `genCPrioridad` es un generador de colas de prioridad. Por ejemplo,

```
ghci> sample genCPrioridad
CP [-4]
CP [-2,-1,-1,2,5]
...
```

```
genCPrioridad :: (Arbitrary a, Num a, Ord a)
                 => Gen (CPrioridad a)
genCPrioridad = do xs <- listOf arbitrary
                   return (foldr inserta vacia xs)

instance (Arbitrary a, Num a, Ord a)
         => Arbitrary (CPrioridad a) where
    arbitrary = genCPrioridad
```

Corrección del generador de colas de prioridad

- Las colas de prioridad producidas por `genCPrioridad` son válidas.

```
prop_genCPrioridad_correcto ::  CPrioridad Int -> Bool
prop_genCPrioridad_correcto c = valida c
```

- Comprobación.

```
ghci> quickCheck prop_genCPrioridad_correcto
+++ OK, passed 100 tests.
```

16.3.3. Especificación de las propiedades de las colas de prioridad

- Si se añade dos elementos a una cola de prioridad se obtiene la misma cola de prioridad idependientemente del orden en que se añadan los elementos.

```
prop_inserta_conmuta :: Int -> Int -> CPrioridad Int
                        -> Bool
prop_inserta_conmuta x y c =
    inserta x (inserta y c) == inserta y (inserta x c)
```

- La cabeza de la cola de prioridad obtenida anadiendo un elemento `x` a la cola de prioridad vacía es `x`.

```
prop_primero_inserta_vacia :: Int -> CPrioridad Int -> Bool
prop_primero_inserta_vacia x c =
    primero (inserta x vacia) == x
```

- El primer elemento de una cola de prioridad `c` no cambia cuando se le añade un elemento mayor o igual que algún elemento de `c`.

```
prop_primero_inserta :: Int -> Int -> CPrioridad Int
                        -> Property
prop_primero_inserta x y c =
    x <= y ==>
    primero (inserta y c') == primero c'
    where c' = inserta x c
```

- El resto de añadir un elemento a la cola de prioridad vacía es la cola vacía.

```
prop_resto_inserta_vacia :: Int -> Bool
prop_resto_inserta_vacia x =
    resto (inserta x vacia) == vacia
```

- El resto de la cola de prioridad obtenida añadiendo un elemento `y` a una cola `c'` (que tiene algún elemento menor o igual que `y`) es la cola que se obtiene añadiendo `y` al resto de `c'`.

```
prop_resto_inserta :: Int -> Int -> CPrioridad Int
                      -> Property
prop_resto_inserta x y c =
    x <= y ==>
```

```
    resto (inserta y c’) == inserta y (resto c’)
    where c’ = inserta x c
```

- vacia es una cola vacía.

```
prop_vacia_es_vacia :: Bool
prop_vacia_es_vacia = esVacia (vacia :: CPrioridad Int)
```

- Si se añade un elemento a una cola de prioridad se obtiene una cola no vacía.

```
prop_inserta_no_es_vacia :: Int -> CPrioridad Int
                            -> Bool
prop_inserta_no_es_vacia x c =
    not (esVacia (inserta x c))
```

16.3.4. Comprobación de las propiedades

Definición del procedimiento de comprobación

- compruebaPropiedades comprueba todas las propiedades con la plataforma de verificación.

```
compruebaPropiedades =
    defaultMain
        [testGroup "Corrección del generador"
          [testProperty "P0" prop_genCPrioridad_correcto],
         testGroup "Propiedade de colas de prioriad:"
          [testProperty "P1" prop_inserta_conmuta,
           testProperty "P2" prop_primero_inserta_vacia,
           testProperty "P3" prop_primero_inserta,
           testProperty "P4" prop_resto_inserta_vacia,
           testProperty "P5" prop_resto_inserta,
           testProperty "P6" prop_vacia_es_vacia,
           testProperty "P7" prop_inserta_no_es_vacia]]
```

Comprobación de las propiedades de las colas de prioridad

```
ghci> compruebaPropiedades
Corrección del generador:
  P0: [OK, passed 100 tests]
```

```
Propiedades de colas de prioridad:
  P1: [OK, passed 100 tests]
  P2: [OK, passed 100 tests]
  P3: [OK, passed 100 tests]
  P4: [OK, passed 100 tests]
  P5: [OK, passed 100 tests]
  P6: [OK, passed 100 tests]
  P7: [OK, passed 100 tests]

         Properties  Total
 Passed  8           8
 Failed  0           0
 Total   8           8
```

Tema 17

El TAD de los conjuntos

17.1. Especificación del TAD de los conjuntos

17.1.1. Signatura del TAD de los conjuntos

- Signatura:

```
vacio,     :: Conj a
inserta    :: Eq a => a -> Conj a -> Conj a
elimina    :: Eq a => a -> Conj a -> Conj a
pertenece  :: Eq a => a -> Conj a -> Bool
esVacio    :: Conj a -> Bool
```

- Descripción de las operaciones:
  - `vacio` es el conjunto vacío.
  - `(inserta x c)` es el conjunto obtenido añadiendo el elemento `x` al conjunto `c`.
  - `(elimina x c)` es el conjunto obtenido eliminando el elemento `x` del conjunto `c`.
  - `(pertenece x c)` se verifica si `x` pertenece al conjunto `c`.
  - `(esVacio c)` se verifica si `c` es el conjunto vacío.

17.1.2. Propiedades del TAD de los conjuntos

1. `inserta x (inserta x c) == inserta x c`
2. `inserta x (inserta y c) == inserta y (inserta x c)`
3. `not (pertenece x vacio)`

4. `pertenece y (inserta x c) == (x==y) || pertenece y c`
5. `elimina x vacio == vacio`
6. Si `x == y`, entonces
   `elimina x (inserta y c) == elimina x c`
7. Si `x /= y`, entonces
   `elimina x (inserta y c) == inserta y (elimina x c)`
8. `esVacio vacio`
9. `not (esVacio (inserta x c))`

17.2. Implementaciones del TAD de los conjuntos

17.2.1. Los conjuntos como listas no ordenadas con duplicados

- Cabecera del módulo:

```
module ConjuntoConListasNoOrdenadasConDuplicados
    (Conj,
     vacio,     -- Conj a
     inserta,   -- Eq a => a -> Conj a -> Conj a
     elimina,   -- Eq a => a -> Conj a -> Conj a
     pertenece, -- Eq a => a -> Conj a -> Bool
     esVacio,   -- Conj a -> Bool
    ) where
```

- El tipo de los conjuntos.

```
newtype Conj a = Cj [a]
```

- Procedimiento de escritura de los conjuntos.

```
instance Show a => Show (Conj a) where
    showsPrec _ (Cj s) cad = showConj s cad

showConj []     cad = showString "{}" cad
showConj (x:xs) cad =
  showChar '{' (shows x (showl xs cad))
  where
```

```
  showl []     cad = showChar '}' cad
  showl (x:xs) cad = showChar ',' (shows x (showl xs cad))
```

- Ejemplo de conjunto: `c1` es el conjunto obtenido añadiéndole al conjunto vacío los elementos 2, 5, 1, 3, 7, 5, 3, 2, 1, 9 y 0.

```
ghci > c1
{2,5,1,3,7,5,3,2,1,9,0}
```

```
c1 :: Conj Int
c1 = foldr inserta vacio [2,5,1,3,7,5,3,2,1,9,0]
```

- `vacio` es el conjunto vacío. Por ejemplo,

```
ghci> vacio
{}
```

```
vacio :: Conj a
vacio = Cj []
```

- `(inserta x c)` es el conjunto obtenido añadiendo el elemento `x` al conjunto `c`. Por ejemplo,

```
c1           ==  {2,5,1,3,7,5,3,2,1,9,0}
inserta 5 c1 ==  {5,2,5,1,3,7,5,3,2,1,9,0}
```

```
inserta :: Eq a => a -> Conj a -> Conj a
inserta x (Cj ys) = Cj (x:ys)
```

- `(elimina x c)` es el conjunto obtenido eliminando el elemento `x` del conjunto `c`. Por ejemplo,

```
c1           ==  {2,5,1,3,7,5,3,2,1,9,0}
elimina 3 c1 ==  {2,5,1,7,5,2,1,9,0}
```

```
elimina :: Eq a => a -> Conj a -> Conj a
elimina x (Cj ys) = Cj (filter (/= x) ys)
```

- `(pertenece x c)` se verifica si `x` pertenece al conjunto `c`. Por ejemplo,

```
c1               ==  {2,5,1,3,7,5,3,2,1,9,0}
pertenece 3 c1   ==  True
pertenece 4 c1   ==  False
```

```
pertenece :: Eq a => a -> Conj a -> Bool
pertenece x (Cj xs) = elem x xs
```

- (`esVacio c`) se verifica si `c` es el conjunto vacío. Por ejemplo,

```
esVacio c1     ⇝  False
esVacio vacio  ⇝  True
```

```
esVacio :: Conj a -> Bool
esVacio (Cj xs) = null xs
```

- (`subconjunto c1 c2`) se verifica si `c1` es un subconjunto de `c2`. Por ejemplo,

```
subconjunto (Cj [1,3,2,1]) (Cj [3,1,3,2])  ⇝  True
subconjunto (Cj [1,3,4,1]) (Cj [3,1,3,2])  ⇝  False
```

```
subconjunto :: Eq a => Conj a -> Conj a -> Bool
subconjunto (Cj xs) (Cj ys) = sublista xs ys
    where sublista [] _      = True
          sublista (x:xs) ys = elem x ys &&
                               sublista xs ys
```

- (`igualConjunto c1 c2`) se verifica si los conjuntos `c1` y `c2` son iguales. Por ejemplo,

```
igualConjunto (Cj [1,3,2,1]) (Cj [3,1,3,2])  ⇝  True
igualConjunto (Cj [1,3,4,1]) (Cj [3,1,3,2])  ⇝  False
```

```
igualConjunto :: Eq a => Conj a -> Conj a -> Bool
igualConjunto c1 c2 =
    subconjunto c1 c2 && subconjunto c2 c1
```

- Los conjuntos son comparables por igualdad.

```
instance Eq a => Eq (Conj a) where
    (==) = igualConjunto
```

17.2.2. Los conjuntos como listas no ordenadas sin duplicados

- Cabecera del módulo.

```
module ConjuntoConListasNoOrdenadasSinDuplicados
    (Conj,
     vacio,     -- Conj a
     esVacio,   -- Conj a -> Bool
     pertenece, -- Eq a => a -> Conj a -> Bool
     inserta,   -- Eq a => a -> Conj a -> Conj a
     elimina    -- Eq a => a -> Conj a -> Conj a
    ) where
```

- Los conjuntos como listas no ordenadas sin repeticiones.

```
newtype Conj a = Cj [a]
```

- Procedimiento de escritura de los conjuntos.

```
instance (Show a) => Show (Conj a) where
    showsPrec _ (Cj s) cad = showConj s cad

showConj []     cad = showString "{}" cad
showConj (x:xs) cad = showChar '{' (shows x (showl xs cad))
  where
  showl []     cad = showChar '}' cad
  showl (x:xs) cad = showChar ',' (shows x (showl xs cad))
```

- Ejemplo de conjunto: `c1` es el conjunto obtenido añadiéndole al conjunto vacío los elementos 2, 5, 1, 3, 7, 5, 3, 2, 1, 9 y 0.

```
ghci> c1
{7,5,3,2,1,9,0}
```

```
c1 :: Conj Int
c1 = foldr inserta vacio [2,5,1,3,7,5,3,2,1,9,0]
```

- `vacio` es el conjunto vacío. Por ejemplo,

```
ghci> vacio
{}
```

```
vacio :: Conj a
vacio = Cj []
```

- (`esVacio c`) se verifica si `c` es el conjunto vacío. Por ejemplo,

```
esVacio c1     ⇝  False
esVacio vacio  ⇝  True
```

```
esVacio :: Conj a -> Bool
esVacio (Cj xs) = null xs
```

- (`pertenece x c`) se verifica si `x` pertenece al conjunto `c`. Por ejemplo,

```
c1              ==  {2,5,1,3,7,5,3,2,1,9,0}
pertenece 3 c1  ==  True
pertenece 4 c1  ==  False
```

```
pertenece :: Eq a => a -> Conj a -> Bool
pertenece x (Cj xs) = elem x xs
```

- (`inserta x c`) es el conjunto obtenido añadiendo el elemento `x` al conjunto `c`. Por ejemplo,

```
inserta 4 c1  ==  {4,7,5,3,2,1,9,0}
inserta 5 c1  ==  {7,5,3,2,1,9,0}
```

```
inserta :: Eq a => a -> Conj a -> Conj a
inserta x s@(Cj xs) | pertenece x s = s
                    | otherwise  = Cj (x:xs)
```

- (`elimina x c`) es el conjunto obtenido eliminando el elemento `x` del conjunto `c`. Por ejemplo,

```
elimina 3 c1  ==  {7,5,2,1,9,0}
```

```
elimina :: Eq a => a -> Conj a -> Conj a
elimina x (Cj ys) = Cj [y | y <- ys, y /= x]
```

- (`subconjunto c1 c2`) se verifica si `c1` es un subconjunto de `c2`. Por ejemplo,

```
subconjunto (Cj [1,3,2]) (Cj [3,1,2])    ⇝  True
subconjunto (Cj [1,3,4,1]) (Cj [1,3,2])  ⇝  False
```

```
subconjunto :: Eq a => Conj a -> Conj a -> Bool
subconjunto (Cj xs) (Cj ys) = sublista xs ys
    where sublista [] _      = True
          sublista (x:xs) ys = elem x ys &&
                               sublista xs ys
```

- (igualConjunto c1 c2) se verifica si los conjuntos c1 y c2 son iguales. Por ejemplo,

```
igualConjunto (Cj [3,2,1]) (Cj [1,3,2])  ⇝  True
igualConjunto (Cj [1,3,4]) (Cj [1,3,2])  ⇝  False
```

```
igualConjunto :: Eq a => Conj a -> Conj a -> Bool
igualConjunto c1 c2 =
    subconjunto c1 c2 && subconjunto c2 c1
```

- Los conjuntos son comparables por igualdad.

```
instance Eq a => Eq (Conj a) where
    (==) = igualConjunto
```

17.2.3. Los conjuntos como listas ordenadas sin duplicados

- Cabecera del módulo

```
module ConjuntoConListasOrdenadasSinDuplicados
    (Conj,
     vacio,     -- Conj a
     esVacio,   -- Conj a -> Bool
     pertenece, -- Ord a => a -> Conj a -> Bool
     inserta,   -- Ord a => a -> Conj a -> Conj a
     elimina    -- Ord a => a -> Conj a -> Conj a
    ) where
```

- Los conjuntos como listas ordenadas sin repeticiones.

```
newtype Conj a = Cj [a]
    deriving Eq
```

- Procedimiento de escritura de los conjuntos.

```
instance (Show a) => Show (Conj a) where
    showsPrec _ (Cj s) cad = showConj s cad

showConj []     cad = showString "{}" cad
showConj (x:xs) cad = showChar '{' (shows x (showl xs cad))
     where showl []     cad = showChar '}' cad
           showl (x:xs) cad = showChar ',' (shows x (showl xs cad))
```

- Ejemplo de conjunto: `c1` es el conjunto obtenido añadiéndole al conjunto vacío los elementos 2, 5, 1, 3, 7, 5, 3, 2, 1, 9 y 0.

```
ghci> c1
{0,1,2,3,5,7,9}
```

```
c1 :: Conj Int
c1 = foldr inserta vacio [2,5,1,3,7,5,3,2,1,9,0]
```

- `vacio` es el conjunto vacío. Por ejemplo,

```
ghci> vacio
{}
```

```
vacio :: Conj a
vacio = Cj []
```

- `(esVacio c)` se verifica si `c` es el conjunto vacío. Por ejemplo,

```
esVacio c1     ⇝  False
esVacio vacio  ⇝  True
```

```
esVacio :: Conj a -> Bool
esVacio (Cj xs) = null xs
```

- `(pertenece x c)` se verifica si `x` pertenece al conjunto `c`. Por ejemplo,

```
c1               ==  {0,1,2,3,5,7,9}
pertenece 3 c1   ==  True
pertenece 4 c1   ==  False
```

```
pertenece :: Ord a => a -> Conj a -> Bool
pertenece x (Cj ys) = elem x (takeWhile (<= x) ys)
```

- `(inserta x c)` es el conjunto obtenido añadiendo el elemento `x` al conjunto `c`. Por ejemplo,

```
c1            ==  {0,1,2,3,5,7,9}
inserta 5 c1  ==  {0,1,2,3,5,7,9}
inserta 4 c1  ==  {0,1,2,3,4,5,7,9}
```

```
inserta :: Ord a => a -> Conj a -> Conj a
inserta x (Cj s) = Cj (agrega x s) where
   agrega x []                   = [x]
   agrega x s@(y:ys) | x > y     = y : (agrega x ys)
                     | x < y     = x : s
                     | otherwise = s
```

- `(elimina x c)` es el conjunto obtenido eliminando el elemento `x` del conjunto `c`. Por ejemplo,

```
c1            ==  {0,1,2,3,5,7,9}
elimina 3 c1  ==  {0,1,2,5,7,9}
```

```
elimina :: Ord a => a -> Conj a -> Conj a
elimina x (Cj s) = Cj (elimina x s) where
   elimina x []                   = []
   elimina x s@(y:ys) | x > y     = y : elimina x ys
                      | x < y     = s
                      | otherwise = ys
```

17.2.4. Los conjuntos de números enteros mediante números binarios

- Los conjuntos que sólo contienen números (de tipo `Int`) entre 0 y $n-1$, se pueden representar como números binarios con n bits donde el bit i $(0 \leq i < n)$ es 1 syss el número i pertenece al conjunto. Por ejemplo,

		43210	
{3,4}	en binario es	11000	en decimal es 24
{1,2,3,4}	en binario es	11110	en decimal es 30
{1,2,4}	en binario es	10110	en decimal es 22

- Cabecera del módulo

```
module ConjuntoConNumerosBinarios
    (Conj,
     vacio,     -- Conj
     esVacio,   -- Conj -> Bool
     pertenece, -- Int -> Conj -> Bool
     inserta,   -- Int -> Conj -> Conj
     elimina    -- Int -> Conj -> Conj
    ) where
```

- Los conjuntos de números enteros como números binarios.

```
newtype Conj = Cj Int deriving Eq
```

- (`conj2Lista c`) es la lista de los elementos del conjunto `c`. Por ejemplo,

```
conj2Lista (Cj 24)  ⇝  [3,4]
conj2Lista (Cj 30)  ⇝  [1,2,3,4]
conj2Lista (Cj 22)  ⇝  [1,2,4]
```

```
conj2Lista (Cj s) = c2l s 0
    where
      c2l 0 _             = []
      c2l n i | odd n     = i : c2l (n `div` 2) (i+1)
              | otherwise = c2l (n `div` 2) (i+1)
```

- Procedimiento de escritura de conjuntos.

```
instance Show Conj where
    showsPrec _ s cad = showConj (conj2Lista s) cad

showConj []     cad = showString "{}" cad
showConj (x:xs) cad =
   showChar '{' (shows x (showl xs cad))
   where
     showl []     cad = showChar '}' cad
     showl (x:xs) cad = showChar ',' (shows x (showl xs cad))
```

- `maxConj` es el máximo número que puede pertenecer al conjunto. Depende de la implementación de Haskell. Por ejemplo,

```
maxConj  ⇝  29
```

```
maxConj :: Int
maxConj =
   truncate (logBase 2 (fromIntegral maxInt)) - 1
   where maxInt = maxBound::Int
```

- Ejemplo de conjunto: `c1` es el conjunto obtenido añadiéndole al conjunto vacío los elementos 2, 5, 1, 3, 7, 5, 3, 2, 1, 9 y 0.

```
ghci> c1
{0,1,2,3,5,7,9}
```

```
c1 :: Conj
c1 = foldr inserta vacio [2,5,1,3,7,5,3,2,1,9,0]
```

- `vacio` es el conjunto vacío. Por ejemplo,

```
ghci> vacio
{}
```

```
vacio :: Conj
vacio = Cj 0
```

- `(esVacio c)` se verifica si `c` es el conjunto vacío. Por ejemplo,

```
esVacio c1     ⇝  False
esVacio vacio  ⇝  True
```

```
esVacio :: Conj -> Bool
esVacio (Cj n) = n == 0
```

- `(pertenece x c)` se verifica si `x` pertenece al conjunto `c`. Por ejemplo,

```
c1                ==  {0,1,2,3,5,7,9}
pertenece 3 c1    ==  True
pertenece 4 c1    ==  False
```

```
pertenece :: Int -> Conj -> Bool
pertenece i (Cj s)
    | (i>=0) && (i<=maxConj) = odd (s `div` (2^i))
    | otherwise
      = error ("pertenece: elemento ilegal =" ++ show i)
```

- (inserta x c) es el conjunto obtenido añadiendo el elemento x al conjunto c. Por ejemplo,

```
c1            ==  {0,1,2,3,5,7,9}
inserta 5 c1  ==  {0,1,2,3,5,7,9}
inserta 4 c1  ==  {0,1,2,3,4,5,7,9}
```

```
inserta i (Cj s)
    | (i>=0) && (i<=maxConj) = Cj (d'*e+m)
    | otherwise
      = error ("inserta: elemento ilegal =" ++ show i)
    where (d,m) = divMod s e
          e     = 2^i
          d'    = if odd d then d else d+1
```

- (elimina x c) es el conjunto obtenido eliminando el elemento x del conjunto c. Por ejemplo,

```
c1            ==  {0,1,2,3,5,7,9}
elimina 3 c1  ==  {0,1,2,5,7,9}
```

```
elimina i (Cj s)
    | (i>=0) && (i<=maxConj) = Cj (d'*e+m)
    | otherwise
      = error ("elimina: elemento ilegal =" ++ show i)
    where (d,m) = divMod s e
          e     = 2^i
          d'    = if odd d then d-1 else d
```

17.3. Comprobación de las implementaciones con QuickCheck

17.3.1. Librerías auxiliares

- Importación de la implementación de los conjuntos que se desea verificar.

```
import ConjuntoConListasNoOrdenadasConDuplicados
-- import ConjuntoConListasNoOrdenadasSinDuplicados
-- import ConjuntoConListasOrdenadasSinDuplicados
```

- Importación de las librerías de comprobación

```
import Test.QuickCheck
import Test.Framework
import Test.Framework.Providers.QuickCheck2
```

17.3.2. Generador de conjuntos

- `genConjunto` es un generador de conjuntos. Por ejemplo,

```
ghci> sample genConjunto
{3,-2,-2,-3,-2,4}
{-8,0,4,6,-5,-2}
{}
```

```
genConjunto :: Gen (Conj Int)
genConjunto = do xs <- listOf arbitrary
                 return (foldr inserta vacio xs)

instance Arbitrary (Conj Int) where
    arbitrary = genConjunto
```

17.3.3. Especificación de las propiedades de los conjuntos

- El número de veces que se añada un elemento a un conjunto no importa.

```
prop_independencia_repeticiones :: Int -> Conj Int
                                   -> Bool
```

```
prop_independencia_repeticiones x c =
    inserta x (inserta x c) == inserta x c
```

- El orden en que se añadan los elementos a un conjunto no importa.

```
prop_independencia_del_orden :: Int -> Int -> Conj Int
                                -> Bool
prop_independencia_del_orden x y c =
    inserta x (inserta y c) == inserta y (inserta x c)
```

- El conjunto vacío no tiene elementos.

```
prop_vacio_no_elementos:: Int -> Bool
prop_vacio_no_elementos x =
    not (pertenece x vacio)
```

- Un elemento pertenece al conjunto obtenido añadiendo `x` al conjunto `c` syss es igual a `x` o pertenece a `c`.

```
prop_pertenece_inserta :: Int -> Int -> Conj Int -> Bool
prop_pertenece_inserta x y c =
    pertenece y (inserta x c) == (x==y) || pertenece y c
```

- Al eliminar cualquier elemento del conjunto vacío se obtiene el conjunto vacío.

```
prop_elimina_vacio :: Int -> Bool
prop_elimina_vacio x =
    elimina x vacio == vacio
```

- El resultado de eliminar x en el conjunto obtenido añadiéndole x al conjunto c es c menos x, si x e y son iguales y es el conjunto obtenido añadiéndole y a c menos x, en caso contrario.

```
prop_elimina_inserta :: Int -> Int -> Conj Int -> Bool
prop_elimina_inserta x y c =
    elimina x (inserta y c)
    == if x == y then elimina x c
       else inserta y (elimina x c)
```

- vacio es vacío.

```
prop_vacio_es_vacio :: Bool
prop_vacio_es_vacio =
    esVacio (vacio :: Conj Int)
```

- Los conjuntos construidos con `inserta` no son vacío.

```
prop_inserta_es_no_vacio :: Int -> Conj Int -> Bool
prop_inserta_es_no_vacio x c =
    not (esVacio (inserta x c))
```

17.3.4. Comprobación de las propiedades

Definición del procedimiento de comprobación

- `compruebaPropiedades` comprueba todas las propiedades con la plataforma de verificación.

```
compruebaPropiedades =
    defaultMain
        [testGroup "Propiedades del TAD conjunto:"
          [testProperty "P1" prop_vacio_es_vacio,
           testProperty "P2" prop_inserta_es_no_vacio,
           testProperty "P3" prop_independencia_repeticiones,
           testProperty "P4" prop_independencia_del_orden,
           testProperty "P5" prop_vacio_no_elementos,
           testProperty "P6" prop_pertenece_inserta,
           testProperty "P7" prop_elimina_vacio,
           testProperty "P8" prop_elimina_inserta]]
```

Comprobación de las propiedades de los conjuntos

```
ghci> compruebaPropiedades
Propiedades del TAD conjunto:
  P1: [OK, passed 100 tests]
  P2: [OK, passed 100 tests]
  P3: [OK, passed 100 tests]
  P4: [OK, passed 100 tests]
  P5: [OK, passed 100 tests]
  P6: [OK, passed 100 tests]
```

```
 P7: [OK, passed 100 tests]
 P8: [OK, passed 100 tests]

        Properties  Total
Passed  8           8
Failed  0           0
Total   8           8
```

Tema 18

El TAD de las tablas

18.1. El tipo predefinido de las tablas (“arrays”)

18.1.1. La clase de los índices de las tablas

- La clase de los índices de las tablas es `Ix`.
- `Ix` se encuentra en la librería `Data.Ix`
- Información de la clase `Ix`:

```
ghci> :info Ix
class (Ord a) => Ix a where
  range :: (a, a) -> [a]
  index :: (a, a) -> a -> Int
  inRange :: (a, a) -> a -> Bool
  rangeSize :: (a, a) -> Int
instance Ix Ordering -- Defined in GHC.Arr
instance Ix Integer -- Defined in GHC.Arr
instance Ix Int -- Defined in GHC.Arr
instance Ix Char -- Defined in GHC.Arr
instance Ix Bool -- Defined in GHC.Arr
instance (Ix a, Ix b) => Ix (a, b)
```

- `(range m n)` es la lista de los índices desde `m` hasta `n`, en el orden del índice. Por ejemplo,

```
range (0,4)        ⇝  [0,1,2,3,4]
range (3,9)        ⇝  [3,4,5,6,7,8,9]
range ('b','f')    ⇝  "bcdef"
```

```
range ((0,0),(1,2))  ⇝  [(0,0),(0,1),(0,2),
                          (1,0),(1,1),(1,2)]
```

- `(index (m,n) i)` es el ordinal del índice `i` dentro del rango `(m,n)`. Por ejemplo,

```
index (3,9) 5              ⇝  2
index ('b','f') 'e'        ⇝  3
index ((0,0),(1,2)) (1,1)  ⇝  4
```

- `(inRange (m,n) i)` se verifica si el índice `i` está dentro del rango limitado por `m` y `n`. Por ejemplo,

```
inRange (0,4) 3              ⇝  True
inRange (0,4) 7              ⇝  False
inRange ((0,0),(1,2)) (1,1)  ⇝  True
inRange ((0,0),(1,2)) (1,5)  ⇝  False
```

- `(rangeSize (m,n))` es el número de elementos en el rango limitado por `m` y `n`. Por ejemplo,

```
rangeSize (3,9)            ⇝  7
rangeSize ('b','f')        ⇝  5
rangeSize ((0,0),(1,2))    ⇝  6
```

18.1.2. El tipo predefinido de las tablas ("arrays")

El tipo predefinido de las tablas ("arrays")

- La librería de las tablas es `Data.Array`.
- Para usar las tablas hay que escribir al principio del fichero

```
import Data.Array
```

- Al importar `Data.Array` también se importa `Data.Ix`.
- `(Array i v)` es el tipo de las tablas con índice en `i` y valores en `v`.

Creación de tablas

- `(array (m,n) ivs)` es la tabla de índices en el rango limitado por `m` y `n` definida por la lista de asociación `ivs` (cuyos elementos son pares de la forma (índice, valor)). Por ejemplo,

```
ghci> array (1,3) [(3,6),(1,2),(2,4)]
array (1,3) [(1,2),(2,4),(3,6)]
ghci> array (1,3) [(i,2*i) | i <- [1..3]]
array (1,3) [(1,2),(2,4),(3,6)]
```

Ejemplos de definiciones de tablas

- (cuadrados n) es un vector de n+1 elementos tal que su elemento i–ésimo es i^2. Por ejemplo,

```
ghci> cuadrados 5
array (0,5) [(0,0),(1,1),(2,4),(3,9),(4,16),(5,25)]
```

```
cuadrados :: Int -> Array Int Int
cuadrados n = array (0,n) [(i,i^2) | i <- [0..n]]
```

- v es un vector con 4 elementos de tipo carácter. Por ejemplo,

```
v :: Array Integer Char
v = array (1,4) [(3,'c'),(2,'a'), (1,'f'), (4,'e')]
```

- m es la matriz con 2 filas y 3 columnas tal que el elemento de la posición (i,j) es el producto de i por j.

```
m :: Array (Int, Int) Int
m = array ((1,1),(2,3)) [((i,j),i*j)) | i<-[1..2],j<-[1..3]]
```

- Una tabla está indefinida si algún índice está fuera de rango.

```
ghci> array (1,4) [(i , i*i) | i <- [1..4]]
array (1,4) [(1,1),(2,4),(3,9),(4,16)]
ghci> array (1,4) [(i , i*i) | i <- [1..5]]
array *** Exception: Error in array index
ghci> array (1,4) [(i , i*i) | i <- [1..3]]
array (1,4) [(1,1),(2,4),(3,9),(4,***
Exception: (Array.!): undefined array element
```

Descomposición de tablas

- `(t ! i)` es el valor del índice `i` en la tabla `t`. Por ejemplo,

```
ghci> v
array (1,4) [(1,'f'),(2,'a'),(3,'c'),(4,'e')]
ghci> v!3
'c'
ghci> m
array ((1,1),(2,3)) [((1,1),1),((1,2),2),((1,3),3),
                     ((2,1),2),((2,2),4),((2,3),6)]
ghci> m!(2,3)
6
```

- `(bounds t)` es el rango de la tabla `t`. Por ejemplo,

```
bounds m  ⇝  ((1,1),(2,3))
```

- `(indices t)` es la lista de los índices de la tabla `t`. Por ejemplo,

```
indices m  ⇝  [(1,1),(1,2),(1,3),(2,1),(2,2),(2,3)]
```

- `(elems t)` es la lista de los elementos de la tabla `t`. Por ejemplo,

```
elems m  ⇝  [1,2,3,2,4,6]
```

- `(assocs t)` es la lista de asociaciones de la tabla `t`. Por ejemplo,

```
ghci> assocs m
[((1,1),1),((1,2),2),((1,3),3),
 ((2,1),2),((2,2),4),((2,3),6)]
```

Modificación de tablas

- `(t // ivs)` es la tabla `t` asignándole a los índices de la lista de asociación `ivs` sus correspondientes valores. Por ejemplo,

```
ghci> m // [((1,1),4), ((2,2),8)]
array ((1,1),(2,3))
      [((1,1),4),((1,2),2),((1,3),3),
       ((2,1),2),((2,2),8),((2,3),6)]
ghci> m
array ((1,1),(2,3))
      [((1,1),1),((1,2),2),((1,3),3),
       ((2,1),2),((2,2),4),((2,3),6)]
```

Definición de tabla por recursión

- (fibs n) es el vector formado por los n primeros términos de la sucesión de Fibonacci. Por ejemplo,

```
ghci> fibs 7
array (0,7) [(0,1),(1,1),(2,2),(3,3),
             (4,5),(5,8),(6,13),(7,21)]
```

```
fibs :: Int -> Array Int Int
fibs n = a where
    a = array (0,n)
              ([(0,1),(1,1)] ++
               [(i,a!(i-1)+a!(i-2)) | i <- [2..n]])
```

Otras funciones de creación de tablas

- (listArray (m,n) vs) es la tabla cuyo rango es (m,n) y cuya lista de valores es vs. Por ejemplo,

```
ghci> listArray (2,5) "Roma"
array (2,5) [(2,'R'),(3,'o'),(4,'m'),(5,'a')]
ghci> listArray ((1,2),(2,4)) [5..12]
array ((1,2),(2,4)) [((1,2),5),((1,3),6),((1,4),7),
                     ((2,2),8),((2,3),9),((2,4),10)]
```

Construcción acumulativa de tablas

- (accumArray f v (m,n) ivs) es la tabla de rango (m,n) tal que el valor del índice i se obtiene acumulando la aplicación de la función f al valor inicial v y a los valores de la lista de asociación ivs cuyo índice es i. Por ejemplo,

```
ghci> accumArray (+) 0 (1,3) [(1,4),(2,5),(1,2)]
array (1,3) [(1,6),(2,5),(3,0)]
ghci> accumArray (*) 1 (1,3) [(1,4),(2,5),(1,2)]
array (1,3) [(1,8),(2,5),(3,1)]
```

- (histograma r is) es el vector formado contando cuantas veces aparecen los elementos del rango r en la lista de índices is. Por ejemplo,

```
ghci> histograma (0,5) [3,1,4,1,5,4,2,7]
array (0,5) [(0,0),(1,2),(2,1),(3,1),(4,2),(5,1)]
```

```
histograma :: (Ix a, Num b) => (a,a) -> [a] -> Array a b
histograma r is =
    accumArray (+) 0 r [(i,1) | i <- is, inRange r i]
```

18.2. Especificación del TAD de las tablas

18.2.1. Signatura del TAD de las tablas

- Signatura:

```
tabla    :: Eq i => [(i,v)] -> Tabla i v
valor    :: Eq i => Tabla i v -> i -> v
modifica :: Eq i => (i,v) -> Tabla i v -> Tabla i v
```

- Descripción de las operaciones:
  - `(tabla ivs)` es la tabla correspondiente a la lista de asociación `ivs` (que es una lista de pares formados por los índices y los valores).
  - `(valor t i)` es el valor del índice `i` en la tabla `t`.
  - `(modifica (i,v) t)` es la tabla obtenida modificando en la tabla `t` el valor de `i` por `v`.

18.2.2. Propiedades del TAD de las tablas

1. `modifica (i,v') (modifica (i,v) t)`
   `= modifica (i,v') t`
2. Si `i /= i'`, entonces
   `modifica (i',v') (modifica (i,v) t)`
   `= modifica (i,v) (modifica (i',v') t)`
3. `valor (modifica (i,v) t) i = v`
4. Si `i /= i'`, entonces
   `valor (modifica (i,v) (modifica (k',v') t)) i'`
   `= valor (modifica (k',v') t) i'`

18.3. Implementaciones del TAD de las tablas

18.3.1. Las tablas como funciones

- Cabecera del módulo:

```
module TablaConFunciones
    (Tabla,
     tabla,   -- Eq i => [(i,v)] -> Tabla i v
     valor,   -- Eq i => Tabla i v -> i -> v
     modifica -- Eq i => (i,v) -> Tabla i v -> Tabla i v
    ) where
```

- Las tablas como funciones.

```
newtype Tabla i v = Tbl (i -> v)
```

- Procedimiento de escritura.

```
instance Show (Tabla i v) where
    showsPrec _ _ cad = showString "<<Una tabla>>" cad
```

- Ejemplos de tablas:

```
t1 = tabla [(i,f i) | i <- [1..6] ]
     where f x | x < 3     = x
               | otherwise = 3-x

t2 = tabla [(4,89), (1,90), (2,67)]
```

- `(valor t i)` es el valor del índice `i` en la tabla `t`. Por ejemplo,

```
valor t1 6  ⇝  -3
valor t2 2  ⇝  67
valor t2 5  ⇝  *** Exception: fuera de rango
```

```
valor :: Eq i => Tabla i v -> i -> v
valor (Tbl f) i = f i
```

- `(modifica (i,v) t)` es la tabla obtenida modificando en la tabla `t` el valor de `i` por `v`. Por ejemplo,

```
valor (modifica (6,9) t1) 6  ⇝  9
```

```
modifica :: Eq i => (i,v) -> Tabla i v -> Tabla i v
modifica (i,v) (Tbl f) = Tbl g
    where g j | j == i    = v
              | otherwise = f j
```

- (`tabla ivs`) es la tabla correspondiente a la lista de asociación `ivs` (que es una lista de pares formados por los índices y los valores). Por ejemplo,

```
ghci> tabla [(4,89), (1,90), (2,67)]
<<Una tabla>>
```

```
tabla :: Eq i => [(i,v)] -> Tabla i v
tabla ivs =
    foldr modifica
          (Tbl (\_ -> error "fuera de rango"))
          ivs
```

18.3.2. Las tablas como listas de asociación

- Cabecera del módulo

```
module TablaConListasDeAsociacion
    (Tabla,
     tabla,   -- Eq i => [(i,v)] -> Tabla i v
     valor,   -- Eq i => Tabla i v -> i -> v
     modifica -- Eq i => (i,v) -> Tabla i v -> Tabla i v
    ) where
```

- Las tablas como listas de asociación.

```
newtype Tabla i v = Tbl [(i,v)]
    deriving Show
```

- Ejemplos de tablas
    - Definición:

```
t1 = tabla [(i,f i) | i <- [1..6] ]
     where f x | x < 3     = x
               | otherwise = 3-x

t2 = tabla [(4,89), (1,90), (2,67)]
```

- Evaluación:

```
ghci> t1
Tbl [(1,1),(2,2),(3,0),(4,-1),(5,-2),(6,-3)]

ghci> t2
Tbl [(4,89),(1,90),(2,67)]
```

- (tabla ivs) es la tabla correspondiente a la lista de asociación ivs (que es una lista de pares formados por los índices y los valores). Por ejemplo,

```
ghci> tabla [(4,89),(1,90),(2,67)]
Tbl [(4,89),(1,90),(2,67)]
```

```
tabla :: Eq i => [(i,v)] -> Tabla i v
tabla ivs = Tbl ivs
```

- (valor t i) es el valor del índice i en la tabla t. Por ejemplo,

```
valor t1 6  ⇝  -3
valor t2 2  ⇝  67
valor t2 5  ⇝  *** Exception: fuera de rango
```

```
valor :: Eq i => Tabla i v -> i -> v
valor (Tbl []) i = error "fuera de rango"
valor (Tbl ((j,v):r)) i
     | i == j    = v
     | otherwise = valor (Tbl r) i
```

- (modifica (i,x) t) es la tabla obtenida modificando en la tabla t el valor de i por x. Por ejemplo,

```
valor t1 6                    ⇝  -3
valor (modifica (6,9) t1) 6  ⇝  9
```

```
modifica :: Eq i => (i,v) -> Tabla i v -> Tabla i v
modifica p (Tbl []) = (Tbl [p])
modifica p'@(i,_) (Tbl (p@(j,_):r))
     | i == j     = Tbl (p':r)
     | otherwise  = Tbl (p:r')
     where Tbl r' = modifica p' (Tbl r)
```

18.3.3. Las tablas como matrices

- Cabecera del módulo:

```
module TablaConMatrices
    (Tabla,
     tabla,     -- Eq i => [(i,v)] -> Tabla i v
     valor,     -- Eq i => Tabla i v -> i -> v
     modifica,  -- Eq i => (i,v) -> Tabla i v -> Tabla i v
     tieneValor -- Ix i => Tabla i v -> i -> Bool
    ) where
```

- Importación de la librería auxiliar:

```
import Data.Array
```

- Las tablas como matrices.

```
newtype Tabla i v = Tbl (Array i v) deriving (Show, Eq)
```

- Ejemplos de tablas:
    - Definición:

```
t1 = tabla [(i,f i) | i <- [1..6] ]
     where f x | x < 3     = x
               | otherwise = 3-x

t2 = tabla [(1,5),(2,4),(3,7)]
```

    - Evaluación:

```
ghci> t1
Tbl (array (1,6) [(1,1),(2,2),(3,0),
                  (4,-1),(5,-2),(6,-3)])
```

```
ghci> t2
Tbl (array (1,3) [(1,5),(2,4),(3,7)])
```

- `(tabla ivs)` es la tabla correspondiente a la lista de asociación `ivs` (que es una lista de pares formados por los índices y los valores). Por ejemplo,

```
ghci> tabla [(1,5),(3,7),(2,4)]
Tbl (array (1,3) [(1,5),(2,4),(3,7)])
```

```
tabla :: Ix i => [(i,v)] -> Tabla i v
tabla ivs = Tbl (array (m,n) ivs)
    where indices = [i | (i,_) <- ivs]
          m       = minimum indices
          n       = maximum indices
```

- `(valor t i)` es el valor del índice `i` en la tabla `t`. Por ejemplo,

```
valor t1 6  ~>  -3
valor t2 2  ~>  67
valor t2 5  ~>  *** Exception: fuera de rango
```

```
valor :: Ix i => Tabla i v -> i -> v
valor (Tbl t) i = t ! i
```

- `(modifica (i,x) t)` es la tabla obtenida modificando en la tabla `t` el valor de `i` por `x`. Por ejemplo,

```
valor t1 6                  ~>  -3
valor (modifica (6,9) t1) 6 ~>  9
```

```
modifica :: Ix i => (i,v) -> Tabla i v -> Tabla i v
modifica p (Tbl t) = Tbl (t // [p])
```

- `(cotas t)` son las cotas de la tabla `t`. Por ejemplo,

```
t2        ~>  Tbl (array (1,3) [(1,5),(2,4),(3,7)])
cotas t2  ~>  (1,3)
```

```
cotas :: Ix i => Tabla i v -> (i,i)
cotas (Tbl t) = bounds t
```

- (tieneValor t x) se verifica si x es una clave de la tabla t. Por ejemplo,

```
tieneValor t2 3  ⇝  True
tieneValor t2 4  ⇝  False
```

```
tieneValor :: Ix i => Tabla i v -> i -> Bool
tieneValor t = inRange (cotas t)
```

18.4. Comprobación de las implementaciones con QuickCheck

18.4.1. Librerías auxiliares

- Importación de la implementación de las tablas que se desea verificar.

```
import TablaConListasDeAsociacion
```

- Importación de las librerías de comprobación.

```
import Test.QuickCheck
import Test.Framework
import Test.Framework.Providers.QuickCheck2
```

18.4.2. Generador de tablas

- genTabla es un generador de tablas. Por ejemplo,

```
ghci> sample genTabla
Tbl [(1,0)]
Tbl [(1,-1)]
Tbl [(1,0),(2,-1),(3,1),(4,1),(5,0)]
```

```
genTabla :: Gen (Tabla Int Int)
genTabla =
    do x <- arbitrary
       xs <- listOf arbitrary
       return (tabla (zip [1..] (x:xs)))

instance Arbitrary (Tabla Int Int) where
    arbitrary = genTabla
```

18.4.3. Especificación de las propiedades de las tablas

- Al modificar una tabla dos veces con la misma clave se obtiene el mismos resultado que modificarla una vez con el último valor.

```
prop_modifica_modifica_1 :: Int -> Int -> Int
                            -> Tabla Int Int -> Bool
prop_modifica_modifica_1 i v v' t =
    modifica (i,v') (modifica (i,v) t)
    == modifica (i,v') t
```

- Al modificar una tabla con dos pares con claves distintas no importa el orden en que se añadan los pares.

```
prop_modifica_modifica_2 :: Int -> Int -> Int -> Int
                            -> Tabla Int Int -> Property
prop_modifica_modifica_2 i i' v v' t =
    i /= i' ==>
    modifica (i',v') (modifica (i,v) t)
    == modifica (i,v) (modifica (i',v') t)
```

- El valor de la clave `i` en la tabla obtenida añadiéndole el par `(i,v)` a la tabla `t` es `v`.

```
prop_valor_modifica_1 :: Int -> Int
                         -> Tabla Int Int -> Bool
prop_valor_modifica_1 i v t =
    valor (modifica (i,v) t) i == v
```

- Sean `i` e `j` dos claves distintas. El valor de la clave `j` en la tabla obtenida añadiéndole el par `(i,v)` a la tabla `t'` (que contiene la clave `j`) es el valor de `j` en `t'`.

```
prop_valor_modifica_2 :: Int -> Int -> Int -> Int
                         -> Tabla Int Int -> Property
prop_valor_modifica_2 i v j v' t =
    i /= j ==>
    valor (modifica (i,v) t') j == valor t' j
    where t' = modifica (j,v') t
```

18.4.4. Comprobación de las propiedades

Definición del procedimiento de comprobación

- `compruebaPropiedades` comprueba todas las propiedades con la plataforma de verificación. Por ejemplo,

```
compruebaPropiedades =
    defaultMain
        [testGroup "Propiedades del TAD tabla"
          [testProperty "P1" prop_modifica_modifica_1,
           testProperty "P2" prop_modifica_modifica_2,
           testProperty "P3" prop_valor_modifica_1,
           testProperty "P4" prop_valor_modifica_2]]
```

Comprobación de las propiedades de las tablas

```
Propiedades del TAD tabla:
  P1: [OK, passed 100 tests]
  P2: [OK, passed 100 tests]
  P3: [OK, passed 100 tests]
  P4: [OK, passed 100 tests]

         Properties  Total
 Passed  4           4
 Failed  0           0
 Total   4           4
```

Tema 19

El TAD de los árboles binarios de búsqueda

19.1. Especificación del TAD de los árboles binarios de búsqueda

19.1.1. Signatura del TAD de los árboles binarios de búsqueda

Descripción de los árboles binarios de búsqueda

- Un árbol binario de búsqueda (ABB) es un árbol binario tal que el valor de cada nodo es mayor que los valores de su subárbol izquierdo y es menor que los valores de su subárbol derecho y, además, ambos subárboles son árboles binarios de búsqueda.
- Por ejemplo, al almacenar los valores de [2,3,4,5,6,8,9] en un ABB se puede obtener los siguientes ABB:

```
    5                     5
  /   \                 /   \
 2     6               3     8
  \     \             / \   / \
   4     8           2   4 6   9
  /       \
 3         9
```

- El objetivo principal de los ABB es reducir el tiempo de acceso a los valores.

Signatura del TAD de los árboles binarios de búsqueda

Signatura:

```
vacio     :: ABB
inserta   :: (Ord a,Show a) => a -> ABB a -> ABB a
elimina   :: (Ord a,Show a) => a -> ABB a -> ABB a
crea      :: (Ord a,Show a) => [a] -> ABB a
menor     :: Ord a => ABB a -> a
elementos :: (Ord a,Show a) => ABB a -> [a]
pertenece :: (Ord a,Show a) => a -> ABB a -> Bool
valido    :: (Ord a,Show a) => ABB a -> Bool
```

Descripción de las operaciones:

- `vacio` es el ABB vacío.
- `(pertenece v a)` se verifica si v es el valor de algún nodo del ABB a.
- `(inserta v a)` es el árbol obtenido añadiendo el valor v al ABB a, si no es uno de sus valores.
- `(crea vs)` es el ABB cuyos valores son vs.
- `(elementos a)` es la lista de los valores de los nodos del ABB en el recorrido inorden.
- `(elimina v a)` es el ABB obtenido eliminando el valor v del ABB a.
- `(menor a)` es el mínimo valor del ABB a.
- `(valido a)` se verifica si a es un ABB correcto.

19.1.2. Propiedades del TAD de los árboles binarios de búsqueda

1. `valido vacio`
2. `valido (inserta v a)`
3. `inserta x a /= vacio`
4. `pertenece x (inserta x a)`
5. `not (pertenece x vacio)`
6. `pertenece y (inserta x a)`
   `== (x == y) || pertenece y a`
7. `valido (elimina v a)`
8. `elimina x (inserta x a) == elimina x a`

9. `valido (crea xs)`
10. `elementos (crea xs) == sort (nub xs)`
11. `pertenece v a == elem v (elementos a)`
12. $\forall$ x $\in$ `elementos a (menor a <= x)`

19.2. Implementación del TAD de los árboles binarios de búsqueda

19.2.1. Los ABB como tipo de dato algebraico

Cabecera del módulo:

```
module ArbolBin
    (ABB,
     vacio,     -- ABB
     inserta,   -- (Ord a,Show a) => a -> ABB a -> ABB a
     elimina,   -- (Ord a,Show a) => a -> ABB a -> ABB a
     crea,      -- (Ord a,Show a) => [a] -> ABB a
     crea',     -- (Ord a,Show a) => [a] -> ABB a
     menor,     -- Ord a => ABB a -> a
     elementos, -- (Ord a,Show a) => ABB a -> [a]
     pertenece, -- (Ord a,Show a) => a -> ABB a -> Bool
     valido     -- (Ord a,Show a) => ABB a -> Bool
    ) where
```

- Los ABB como tipo de dato algebraico.

```
data Ord a => ABB a = Vacio
                    | Nodo a (ABB a) (ABB a)
                    deriving (Show, Eq)
```

- Procedimiento de escritura de árboles binarios de búsqueda.

```
instance (Show a, Ord a) => Show (ABB a) where
    show Vacio        = " -"
    show (Nodo x i d) =
       " (" ++ show x ++ show i ++ show d ++ ")"
```

- abb1 y abb2 son árboles de búsqueda binarios.

```
ghci> abb1
 (5 (2 - (4 (3 - -) -)) (6 - (8 - (9 - -))))
ghci> abb2
 (5 (2 - (4 (3 - -) -)) (8 (6 - (7 - -)) (10 (9 - -) (11 - -))))
```

```
abb1, abb2 :: ABB Int
abb1 = crea (reverse [5,2,6,4,8,3,9])
abb2 = foldr inserta vacio
             (reverse [5,2,4,3,8,6,7,10,9,11])
```

- vacio es el ABB vacío.

```
vacio :: ABB a
vacio = Vacio
```

- (pertenece v a) se verifica si v es el valor de algún nodo del ABB a. Por ejemplo,

```
pertenece 3 abb1  ⇝  True
pertenece 7 abb1  ⇝  False
```

```
pertenece :: (Ord a,Show a) => a -> ABB a -> Bool
pertenece v' Vacio                 = False
pertenece v' (Nodo v i d) | v==v' = True
                          | v'<v  = pertenece v' i
                          | v'>v  = pertenece v' d
```

- (inserta v a) es el árbol obtenido añadiendo el valor v al ABB a, si no es uno de sus valores. Por ejemplo,

```
ghci> inserta 7 abb1
 (5 (2 - (4 (3 - -) -)) (6 - (8 (7 - -) (9 - -))))
```

```
inserta :: (Ord a,Show a) => a -> ABB a -> ABB a
inserta v' Vacio = Nodo v' Vacio Vacio
inserta v' (Nodo v i d)
    | v' == v   = Nodo v i d
    | v' < v    = Nodo v (inserta v' i) d
    | otherwise = Nodo v i (inserta v' d)
```

- (crea vs) es el ABB cuyos valores son vs. Por ejemplo

```
ghci> crea [3,7,2]
 (2 - (7 (3 - -) -))
```

```
crea :: (Ord a,Show a) => [a] -> ABB a
crea = foldr inserta Vacio
```

- (crea' vs) es el ABB de menor profundidad cuyos valores son los de la lista ordenada vs. Por ejemplo,

```
ghci> crea' [2,3,7]
 (3 (2 - -) (7 - -))
```

```
crea' :: (Ord a,Show a) => [a] -> ABB a
crea' [] = Vacio
crea' vs = Nodo x (crea' l1) (crea' l2)
    where n      = length vs `div` 2
          l1     = take n vs
          (x:l2) = drop n vs
```

- (elementos a) es la lista de los valores de los nodos del ABB a en el recorrido inorden. Por ejemplo,

```
elementos abb1  ⇝  [2,3,4,5,6,8,9]
elementos abb2  ⇝  [2,3,4,5,6,7,8,9,10,11]
```

```
elementos :: (Ord a,Show a) => ABB a -> [a]
elementos Vacio = []
elementos (Nodo v i d) =
   elementos i ++ [v] ++ elementos d
```

- (elimina v a) el ABB obtenido eliminando el valor v del ABB a.

```
ghci> elimina 3 abb1
 (5 (2 - (4 - -)) (6 - (8 - (9 - -))))
ghci> elimina 2 abb1
 (5 (4 (3 - -) -) (6 - (8 - (9 - -))))
```

```
elimina  :: (Ord a,Show a) => a -> ABB a -> ABB a
elimina v' Vacio = Vacio
elimina v' (Nodo v i Vacio) | v'==v = i
elimina v' (Nodo v Vacio d) | v'==v = d
elimina v' (Nodo v i d)
    | v'<v  = Nodo v (elimina v' i) d
    | v'>v  = Nodo v i (elimina v' d)
    | v'==v = Nodo k i (elimina k d)
            where k = menor d
```

- (`menor a`) es el mínimo valor del ABB a. Por ejemplo,

```
menor abb1  ⇝  2
```

```
menor :: Ord a => ABB a -> a
menor (Nodo v Vacio _) = v
menor (Nodo _ i _)        = menor i
```

- (`menorTodos v a`) se verifica si v es menor que todos los elementos del ABB a.

```
menorTodos :: (Ord a, Show a) => a -> ABB a -> Bool
menorTodos v Vacio = True
menorTodos v a        = v < minimum (elementos a)
```

- (`mayorTodos v a`) se verifica si v es mayor que todos los elementos del ABB a.

```
mayorTodos :: (Ord a, Show a) => a -> ABB a -> Bool
mayorTodos v Vacio = True
mayorTodos v a = v > maximum (elementos a)
```

- (`valido a`) se verifica si a es un ABB correcto. Por ejemplo,

```
valido abb1 ⇝ True
```

```
valido :: (Ord a, Show a) => ABB a -> Bool
valido Vacio        = True
valido (Nodo v i d) = mayorTodos v i && menorTodos v d
                      && valido i && valido d
```

19.3. Comprobación de la implementación con QuickCheck

19.3.1. Librerías auxiliares

- Importación de la implementación de ABB.

```
import ArbolBin
```

- Importación de librerías auxiliares.

```
import Data.List
import Test.QuickCheck
import Test.Framework
import Test.Framework.Providers.QuickCheck2
```

19.3.2. Generador de árboles binarios de búsqueda

- `genABB` es un generador de árboles binarios de búsqueda. Por ejemplo,

```
ghci> sample genABB
 -
 (1 (-1 - -) -)
 (1 - -)
 (-1 (-3 - -) (1 - (4 - -)))
```

```
genABB :: Gen (ABB Int)
genABB = do xs <- listOf arbitrary
            return (foldr inserta vacio xs)

instance Arbitrary (ABB Int) where
    arbitrary = genABB
```

- Propiedad. Todo los elementos generados por genABB son árboles binarios de búsqueda.

```
prop_genABB_correcto :: ABB Int -> Bool
prop_genABB_correcto = valido
```

- `listaOrdenada` es un generador de listas ordenadas de números enteros. Por ejemplo,

```
ghci> sample listaOrdenada
[1]
[-2,-1,0]
```

```
listaOrdenada :: Gen [Int]
listaOrdenada =
    frequency [(1,return []),
               (4,do xs <- orderedList
                     n <- arbitrary
                     return (nub ((case xs of
                                    []  -> n
                                    x:_ -> n `min` x)
                                  :xs)))]
```

- (`ordenada xs`) se verifica si `xs` es una lista ordenada creciente. Por ejemplo,

```
ordenada [3,5,9]  ⇝  True
ordenada [3,9,5]  ⇝  False
```

```
ordenada :: [Int] -> Bool
ordenada xs = and [x<y | (x,y) <- zip xs (tail xs)]
```

- Propiedad. El generador `listaOrdenada` produce listas ordenadas.

```
prop_listaOrdenada_correcta :: [Int] -> Property
prop_listaOrdenada_correcta xs =
    forAll listaOrdenada ordenada
```

19.3.3. Especificación de las propiedades de los árboles de búsqueda

- `vacio` es un ABB.

```
prop_vacio_es_ABB :: Bool
prop_vacio_es_ABB =
    valido (vacio :: ABB Int)
```

- Si a es un ABB, entonces (`inserta v a`) también lo es.

```
prop_inserta_es_valida :: Int -> ABB Int -> Bool
prop_inserta_es_valida v a =
    valido (inserta v a)
```

- El árbol que resulta de añadir un elemento a un ABB es no vacío.

```
prop_inserta_es_no_vacio :: Int -> ABB Int -> Bool
prop_inserta_es_no_vacio x a =
    inserta x a /= vacio
```

- Para todo x y a, x es un elemento de (inserta x a).

```
prop_elemento_de_inserta :: Int -> ABB Int -> Bool
prop_elemento_de_inserta x a =
    pertenece x (inserta x a)
```

- En un árbol vacio no hay ningún elemento.

```
prop_vacio_sin_elementos :: Int -> Bool
prop_vacio_sin_elementos x =
    not (pertenece x vacio)
```

- Los elementos de (inserta x a) son x y los elementos de a.

```
prop_elementos_de_inserta :: Int -> Int
                             -> ABB Int -> Bool
prop_elementos_de_inserta x y a =
    pertenece y (inserta x a)
    == (x == y) || pertenece y a
```

- Si a es un ABB, entonces (elimina v a) también lo es.

```
prop_elimina_es_valida :: Int -> ABB Int -> Bool
prop_elimina_es_valida v a =
    valido (elimina v a)
```

- El resultado de eliminar el elemento x en (inserta x a) es (elimina x a).

```
prop_elimina_agrega :: Int -> ABB Int -> Bool
prop_elimina_agrega x a =
    elimina (inserta x a) == elimina x a
```

- (crea xs) es un ABB.

```
prop_crea_es_valida :: [Int] -> Bool
prop_crea_es_valida xs =
    valido (crea xs)
```

- Para todas las listas ordenadas xs, se tiene que (crea' xs) es un ABB.

```
prop_crea'_es_valida :: [Int] -> Property
prop_crea'_es_valida xs =
    forAll listaOrdenada (valido . crea')
```

- (elementos (crea xs)) es igual a la lista xs ordenada y sin repeticiones.

```
prop_elementos_crea :: [Int] -> Bool
prop_elementos_crea xs =
    elementos (crea xs) == sort (nub xs)
```

- Si ys es una lista ordenada sin repeticiones, entonces (elementos (crea' ys)) es igual ys.

```
prop_elementos_crea' :: [Int] -> Bool
prop_elementos_crea' xs =
    elementos (crea' ys) == ys
    where ys = sort (nub xs)
```

- Un elemento pertenece a (elementos a) syss es un valor de a.

```
prop_en_elementos :: Int -> ABB Int -> Bool
prop_en_elementos v a =
    pertenece v a == elem v (elementos a)
```

- (menor a) es menor o igual que todos los elementos de ABB a.

```
prop_menoresMinimo ::Int -> ABB Int -> Bool
prop_menoresMinimo v a =
    and [menor a <= v | v <- elementos a]
```

19.3.4. Comprobación de las propiedades

Definición del procedimiento de comprobación

- `compruebaPropiedades` comprueba todas las propiedades con la plataforma de verificación.

```
compruebaPropiedades =
    defaultMain
        [testGroup "Propiedades del tipo ABB"
          [testProperty "P1"  prop_listaOrdenada_correcta,
           testProperty "P2"  prop_orderedList_correcta,
           testProperty "P3"  prop_vacio_es_ABB,
           testProperty "P4"  prop_inserta_es_valida,
           testProperty "P5"  prop_inserta_es_no_vacio,
           testProperty "P6"  prop_elemento_de_inserta,
           testProperty "P7"  prop_vacio_sin_elementos,
           testProperty "P8"  prop_elementos_de_inserta,
           testProperty "P9"  prop_elimina_es_valida,
           testProperty "P10" prop_elimina_agrega,
           testProperty "P11" prop_crea_es_valida,
           testProperty "P12" prop_crea'_es_valida,
           testProperty "P13" prop_elementos_crea,
           testProperty "P14" prop_elementos_crea',
           testProperty "P15" prop_en_elementos,
           testProperty "P16" prop_menoresMinimo],
         testGroup "Corrección del generador"
          [testProperty "P18" prop_genABB_correcto]]
```

Comprobación de las propiedades de los ABB

```
ghci> compruebaPropiedades
Propiedades del tipo ABB:
  P1: [OK, passed 100 tests]
  P2: [OK, passed 100 tests]
  P3: [OK, passed 100 tests]
  P4: [OK, passed 100 tests]
  P5: [OK, passed 100 tests]
  P6: [OK, passed 100 tests]
  P7: [OK, passed 100 tests]
  P8: [OK, passed 100 tests]
```

```
  P9: [OK, passed 100 tests]
  P10: [OK, passed 100 tests]
  P11: [OK, passed 100 tests]
  P12: [OK, passed 100 tests]
  P13: [OK, passed 100 tests]
  P14: [OK, passed 100 tests]
  P15: [OK, passed 100 tests]
  P16: [OK, passed 100 tests]
Corrección del generador:
  P18: [OK, passed 100 tests]

         Properties   Total
 Passed  17           17
 Failed  0            0
 Total   17           17
```

Tema 20

El TAD de los montículos

20.1. Especificación del TAD de los montículos

20.1.1. Signatura del TAD de los montículos

Descripción de los montículos

Un montículo es un árbol binario en el que los valores de cada nodo es menor o igual que los valores de sus hijos. Por ejemplo,

```
        1              1
       / \            / \
      /   \          /   \
     2     6        3     6
    / \   / \      / \   / \
   3   8 9   7    4   2 9   7
```

el de la izquierda es un montículo, pero el de la derecha no lo es.

Signatura del TAD de los montículos

Signatura:

```
vacio   :: Ord a => Monticulo a
inserta :: Ord a => a -> Monticulo a -> Monticulo a
menor   :: Ord a => Monticulo a -> a
resto   :: Ord a => Monticulo a -> Monticulo a
esVacio :: Ord a => Monticulo a -> Bool
valido  :: Ord a => Monticulo a -> Bool
```

Descripción de las operaciones:

- `vacio` es el montículo vacío.

- `(inserta x m)` es el montículo obtenido añadiendo el elemento `x` al montículo `m`.
- `(menor m)` es el menor elemento del montículo `m`.
- `(resto m)` es el montículo obtenido eliminando el menor elemento del montículo `m`.
- `(esVacio m)` se verifica si `m` es el montículo vacío.
- `(valido m)` se verifica si `m` es un montículo; es decir, es un árbol binario en el que los valores de cada nodo es menor o igual que los valores de sus hijos.

20.1.2. Propiedades del TAD de los montículos

1. `esVacio vacio`
2. `valido (inserta x m)`
3. `not (esVacio (inserta x m))`
4. `not (esVacio m) ==> valido (resto m)`
5. `resto (inserta x vacio) == vacio`
6. `x <= menor m ==> resto (inserta x m) == m`
7. Si `m` es no vacío y `x > menor m`, entonces
   `resto (inserta x m) == inserta x (resto m)`
8. `esVacio m ||`
   `esVacio (resto m) ||`
   `menor m <= menor (resto m)`

20.2. Implementación del TAD de los montículos

20.2.1. Los montículos como tipo de dato algebraico

- Cabecera del módulo:

```
module Monticulo
    (Monticulo,
     vacio,   -- Ord a => Monticulo a
     inserta, -- Ord a => a -> Monticulo a -> Monticulo a
     menor,   -- Ord a => Monticulo a -> a
     resto,   -- Ord a => Monticulo a -> Monticulo a
```

```
    esVacio, -- Ord a => Monticulo a -> Bool
    valido   -- Ord a => Monticulo a -> Bool
   ) where
```

- Librería auxiliar:

```
import Data.List (sort)
```

- Los montículos como tipo de dato algebraico

```
data Ord a => Monticulo a
                = Vacio
                | M a Int (Monticulo a) (Monticulo a)
                deriving Show
```

- La forma de los montículos no vacío es (M v r i d) donde
  - v es el valor de la raíz del montículo.
  - r es el rango del montículo; es decir, la menor distancia de la raíz a un montículo vacío.
  - i es el submontículo izquierdo y
  - f es el submontículo derecho.

Ejemplos de montículos

- Definición:

```
m1, m2, m3 :: Monticulo Int
m1 = foldr inserta vacio [6,1,4,8]
m2 = foldr inserta vacio [7,5]
m3 = mezcla m1 m2
```

- Representación:

```
        m1              m2                 m3
                                          (1,2)
        (1,2)           (5,1)            /     \
       /     \         /                /       \
    (4,1)   (6,1)  (7,1)            (5,2)        (4,1)
    /                               /    \        /
(8,1)                           (7,1)   (6,1)  (8,1)
```

- `vacio` es el montículo vacío.

```
vacio :: Ord a => Monticulo a
vacio = Vacio
```

- `(rango m)` es el rango del montículo `m`; es decir, la menor distancia a un montículo vacío. Por ejemplo,

```
rango m1  ⇝  2
rango m2  ⇝  1
```

```
rango :: Ord a => Monticulo a -> Int
rango Vacio       = 0
rango (M _ r _ _) = r
```

`(creaM x a b)` es el montículo creado a partir del elemento `x` y los montículos `a` y `b`. Se supone que `x` es menor o igual que el mínimo de `a` y de `b`. Por ejemplo,

```
ghci> m1
M 1 2 (M 4 1 (M 8 1 Vacio Vacio) Vacio) (M 6 1 Vacio Vacio)
ghci> m2
M 5 1 (M 7 1 Vacio Vacio) Vacio
ghci> creaM 0 m1 m2
M 0 2 (M 1 2 (M 4 1 (M 8 1 Vacio Vacio) Vacio) (M 6 1 Vacio Vacio))
      (M 5 1 (M 7 1 Vacio Vacio) Vacio)
```

```
creaM :: Ord a => a -> Monticulo a -> Monticulo a -> Monticulo a
creaM x a b | rango a >= rango b = M x (rango b + 1) a b
            | otherwise          = M x (rango a + 1) b a
```

- `(mezcla m1 m2)` es el montículo obtenido mezclando los montículos `m1` y `m2`. Por ejemplo,

```
ghci> mezcla m1 m2
M 1 2 (M 5 2 (M 7 1 Vacio Vacio) (M 6 1 Vacio Vacio))
      (M 4 1 (M 8 1 Vacio Vacio) Vacio)
```

```
mezcla :: Ord a =>  Monticulo a -> Monticulo a
                    -> Monticulo a
mezcla m Vacio = m
mezcla Vacio m = m
```

```
mezcla m1@(M x _ a1 b1) m2@(M y _ a2 b2)
      | x <= y     = creaM x a1 (mezcla b1 m2)
      | otherwise = creaM y a2 (mezcla m1 b2)
```

- `(inserta x m)` es el montículo obtenido añadiendo el elemento `x` al montículo `m`. Por ejemplo,

```
ghci> m1
M 1 2 (M 4 1 (M 8 1 Vacio Vacio) Vacio)
      (M 6 1 Vacio Vacio)
ghci> inserta 3 m1
M 1 2
  (M 4 1 (M 8 1 Vacio Vacio) Vacio)
  (M 3 1 (M 6 1 Vacio Vacio) Vacio)
```

```
inserta :: Ord a => a -> Monticulo a -> Monticulo a
inserta x m = mezcla (M x 1 Vacio Vacio) m
```

- `(menor m)` es el menor elemento del montículo `m`. Por ejemplo,

```
menor m1  ⇝  1
menor m2  ⇝  5
```

```
menor  :: Ord a => Monticulo a -> a
menor (M x _ _ _) = x
menor Vacio       = error "menor: monticulo vacio"
```

- `(resto m)` es el montículo obtenido eliminando el menor elemento del montículo `m`. Por ejemplo,

```
ghci> resto m1
M 4 2 (M 8 1 Vacio Vacio) (M 6 1 Vacio Vacio)
```

```
resto :: Ord a => Monticulo a -> Monticulo a
resto Vacio       = error "resto: monticulo vacio"
resto (M x _ a b) = mezcla a b
```

- `(esVacio m)` se verifica si m es el montículo vacío.

```
esVacio :: Ord a => Monticulo a -> Bool
esVacio Vacio = True
esVacio _     = False
```

- (`valido m`) se verifica si `m` es un montículo; es decir, es un árbol binario en el que los valores de cada nodo es menor o igual que los valores de sus hijos. Por ejemplo,

```
valido m1  ⇝  True
valido (M 3 5 (M 2 1 Vacio Vacio) Vacio)  ⇝  False
```

```
valido :: Ord a => Monticulo a -> Bool
valido Vacio = True
valido (M x _ Vacio Vacio) = True
valido (M x _ m1@(M x1 n1 a1 b1) Vacio) =
    x <= x1 && valido m1
valido (M x _ Vacio m2@(M x2 n2 a2 b2)) =
    x <= x2 && valido m2
valido (M x _ m1@(M x1 n1 a1 b1) m2@(M x2 n2 a2 b2)) =
    x <= x1 && valido m1 &&
    x <= x2 && valido m2
```

- (`elementos m`) es la lista de los elementos del montículo `m`. Por ejemplo,

```
elementos m1  ⇝  [1,4,8,6]
```

```
elementos :: Ord a => Monticulo a -> [a]
elementos Vacio       = []
elementos (M x _ a b) = x : elementos a ++ elementos b
```

- (`equivMonticulos m1 m2`) se verifica si los montículos `m1` y `m2` tienen los mismos elementos. Por ejemplo,

```
ghci> m1
M 1 2 (M 4 1 (M 8 1 Vacio Vacio) Vacio)
      (M 6 1 Vacio Vacio)
ghci> let m1' = foldr inserta vacio [6,8,4,1]
M 1 2 (M 4 1 Vacio Vacio)
      (M 6 1 (M 8 1 Vacio Vacio) Vacio)
ghci> equivMonticulos m1 m1'
True
```

```
equivMonticulos :: Ord a => Monticulo a -> Monticulo a
                   -> Bool
equivMonticulos m1 m2 =
    sort (elementos m1) == sort (elementos m2)
```

- Los montículos son comparables por igualdad.

```
instance Ord a => Eq (Monticulo a) where
   (==) = equivMonticulos
```

20.3. Comprobación de la implementación con QuickCheck

20.3.1. Librerías auxiliares

- Importación de la implementación a verificar.

```
import Monticulo
```

- Importación de librerías auxiliares.

```
import Test.QuickCheck
import Test.Framework
import Test.Framework.Providers.QuickCheck2
```

20.3.2. Generador de montículos

- `(creaMonticulo xs)` es el montículo correspondiente a la lista `xs`. Por ejemplo,

```
ghci> creaMonticulo [6,1,4,8]
M 1 2 (M 4 1 (M 8 1 Vacio Vacio) Vacio)
      (M 6 1 Vacio Vacio)
ghci> creaMonticulo [6,8,4,1]
M 1 2 (M 4 1 Vacio Vacio)
      (M 6 1 (M 8 1 Vacio Vacio) Vacio)
```

```
creaMonticulo :: [Int] -> Monticulo Int
creaMonticulo = foldr inserta vacio
```

- `genMonticulo` es un generador de montículos. Por ejemplo,

```
ghci> sample genMonticulo
VacioM
M (-1) 1 (M 1 1 VacioM VacioM) VacioM
...
```

```
genMonticulo :: Gen (Monticulo Int)
genMonticulo = do xs <- listOf arbitrary
                  return (creaMonticulo xs)

instance Arbitrary (Monticulo Int) where
    arbitrary = genMonticulo
```

Corrección del generador de montículos

- Prop.: `genMonticulo` genera montículos válidos.

```
prop_genMonticulo :: Monticulo Int -> Bool
prop_genMonticulo m = valido m
```

Comprobación:

```
ghci> quickCheck prop_genMonticulo
+++ OK, passed 100 tests.
```

Generador de montículos no vacíos

- `monticuloNV` es un generador de montículos no vacío. Por ejemplo,

```
ghci> sample monticuloNV
M 0 1 VacioM VacioM
M 1 1 (M 1 1 (M 1 1 VacioM VacioM) VacioM) VacioM
...
```

```
monticuloNV :: Gen (Monticulo Int)
monticuloNV = do xs <- listOf arbitrary
                 x <- arbitrary
                 return (creaMonticulo (x:xs))
```

Corrección del generador de montículos no vacíos

- Prop.: `monticuloNV` genera montículos no vacío.

```
prop_monticuloNV :: Monticulo Int -> Property
prop_monticuloNV m =
    forAll monticuloNV
           (\m -> (valido m) && not (esVacio m))
```

Comprobación:

```
ghci> quickCheck prop_monticuloNV
+++ OK, passed 100 tests.
```

20.3.3. Especificación de las propiedades de los montículos

- `vacio` es un montículo.

```
prop_vacio_es_monticulo :: Bool
prop_vacio_es_monticulo =
    esVacio (vacio :: Monticulo Int)
```

- `inserta` produce montículos válidos.

```
prop_inserta_es_valida :: Int -> Monticulo Int -> Bool
prop_inserta_es_valida x m =
    valido (inserta x m)
```

- Los montículos creados con `inserta` son no vacío.

```
prop_inserta_no_vacio :: Int -> Monticulo Int -> Bool
prop_inserta_no_vacio x m =
    not (esVacio (inserta x m))
```

- Al borrar el menor elemento de un montículo no vacío se obtiene un montículo válido.

```
prop_resto_es_valida :: Monticulo Int -> Property
prop_resto_es_valida m =
    forAll monticuloNV  (\m -> valido (resto m))
```

- El resto de (inserta x m) es m si m es el montículo vacío o x es menor o igual que el menor elemento de m y es (inserta x (resto m)), en caso contrario.

```
prop_resto_inserta :: Int -> Monticulo Int -> Bool
prop_resto_inserta x m =
    resto (inserta x m)
    == if esVacio m || x <= menor m then m
       else inserta x (resto m)
```

- (menor m) es el menor elemento del montículo m.

```
prop_menor_es_minimo :: Monticulo Int -> Bool
prop_menor_es_minimo m =
    esVacio m || esVacio (resto m) ||
    menor m <= menor (resto m)
```

20.3.4. Comprobación de las propiedades

Definición del procedimiento de comprobación

- compruebaPropiedades comprueba todas las propiedades con la plataforma de verificación.

```
compruebaPropiedades =
    defaultMain
        [testGroup "Propiedades del TAD monticulo"
         [testProperty "P1" prop_genMonticulo,
          testProperty "P2" prop_monticuloNV,
          testProperty "P3" prop_vacio_es_monticulo,
          testProperty "P4" prop_inserta_es_valida,
          testProperty "P5" prop_inserta_no_vacio,
          testProperty "P6" prop_resto_es_valida,
          testProperty "P7" prop_resto_inserta,
          testProperty "P8" prop_menor_es_minimo]]
```

Comprobación de las propiedades de los montículos

```
ghci> compruebaPropiedades
Propiedades del TAD monticulo:
  P1: [OK, passed 100 tests]
  P2: [OK, passed 100 tests]
```

```
 P3: [OK, passed 100 tests]
 P4: [OK, passed 100 tests]
 P5: [OK, passed 100 tests]
 P6: [OK, passed 100 tests]
 P7: [OK, passed 100 tests]
 P8: [OK, passed 100 tests]

        Properties  Total
Passed  8           8
Failed  0           0
Total   8           8
```

20.4. Implementación de las colas de prioridad mediante montículos

20.4.1. Las colas de prioridad como montículos

Cabecera del módulo:

```
module ColaDePrioridadConMonticulos
    (CPrioridad,
     vacia,   -- Ord a => CPrioridad a
     inserta, -- Ord a => a -> CPrioridad a -> CPrioridad a
     primero, -- Ord a => CPrioridad a -> a
     resto,   -- Ord a => CPrioridad a -> CPrioridad a
     esVacia, -- Ord a => CPrioridad a -> Bool
     valida   -- Ord a => CPrioridad a -> Bool
    ) where
```

Importación cualificada:

```
import qualified Monticulo as M
```

- Descripción de las operaciones:
  - `vacia` es la cola de prioridad vacía.
  - `(inserta x c)` añade el elemento `x` a la cola de prioridad `c`.
  - `(primero c)` es el primer elemento de la cola de prioridad `c`.
  - `(resto c)` es el resto de la cola de prioridad `c`.

  - (`esVacia c`) se verifica si la cola de prioridad `c` es vacía.
  - (`valida c`) se verifica si `c` es una cola de prioridad válida.

- Las colas de prioridad como montículos.

```
newtype CPrioridad a = CP (M.Monticulo a)
    deriving (Eq, Show)
```

- Ejemplo de cola de prioridad:

```
cp1 :: CPrioridad Int
cp1 = foldr inserta vacia [3,1,7,2,9]
```

- Evaluación:

```
ghci> cp1
CP (M 1 2
      (M 2 2
         (M 9 1 VacioM VacioM)
         (M 7 1 VacioM VacioM))
      (M 3 1 VacioM VacioM))
```

- `vacia` es la cola de prioridad vacía. Por ejemplo,

```
vacia  ⇝ CP Vacio
```

```
vacia :: Ord a => CPrioridad a
vacia = CP M.vacio
```

- (`inserta x c`) añade el elemento `x` a la cola de prioridad `c`. Por ejemplo,

```
ghci> inserta 5 cp1
CP (M 1 2
      (M 2 2
         (M 9 1 VacioM VacioM)
         (M 7 1 VacioM VacioM))
      (M 3 1
         (M 5 1 VacioM VacioM) VacioM))
```

```
inserta :: Ord a => a -> CPrioridad a -> CPrioridad a
inserta v (CP c) = CP (M.inserta v c)
```

- (primero c) es la cabeza de la cola de prioridad c. Por ejemplo,

```
primero cp1  ⇝  1
```

```
primero :: Ord a => CPrioridad a -> a
primero (CP c) = M.menor c
```

- (resto c) elimina la cabeza de la cola de prioridad c. Por ejemplo,

```
ghci> resto cp1
CP (M 2 2
      (M 9 1 VacioM VacioM)
      (M 3 1
         (M 7 1 VacioM VacioM) VacioM))
```

```
resto :: Ord a => CPrioridad a -> CPrioridad a
resto (CP c) = CP (M.resto c)
```

- (esVacia c) se verifica si la cola de prioridad c es vacía. Por ejemplo,

```
esVacia cp1    ⇝  False
esVacia vacia  ⇝  True
```

```
esVacia :: Ord a => CPrioridad a -> Bool
esVacia (CP c) = M.esVacio c
```

- (valida c) se verifica si c es una cola de prioridad válida. En la representación mediante montículo todas las colas de prioridad son válidas.

```
valida :: Ord a => CPrioridad a -> Bool
valida _ = True
```

Tema 21

El TAD de los polinomios

21.1. Especificación del TAD de los polinomios

21.1.1. Signatura del TAD de los polinomios

Signatura del TAD de los polinomios

Signatura:

```
polCero   :: Polinomio a
esPolCero :: Num a =>  Polinomio a -> Bool
consPol   :: Num a => Int -> a -> Polinomio a -> Polinomio a
grado     :: Polinomio a -> Int
coefLider :: Num a => Polinomio a -> a
restoPol  :: Polinomio a -> Polinomio a
```

Descripción de las operaciones:

- `polCero` es el polinomio cero.
- `(esPolCero p)` se verifica si `p` es el polinomio cero.
- `(consPol n b p)` es el polinomio $bx^n + p$.
- `(grado p)` es el grado del polinomio `p`.
- `(coefLider p)` es el coeficiente líder del polinomio `p`.
- `(restoPol p)` es el resto del polinomio `p`.

Ejemplos de polinomios

Ejemplos de polinomios que se usarán en lo sucesivo.

- Definición:

```
ejPol1, ejPol2, ejPol3, ejTerm:: Polinomio Int
ejPol1 = consPol 4 3 (consPol 2 (-5) (consPol 0 3 polCero))
ejPol2 = consPol 5 1 (consPol 2 5 (consPol 1 4 polCero))
ejPol3 = consPol 4 6 (consPol 1 2 polCero)
ejTerm = consPol 1 4 polCero
```

- Evaluación:

```
ejPol1  ⇝  3*x^4 + -5*x^2 + 3
ejPol2  ⇝  x^5 + 5*x^2 + 4*x
ejPol3  ⇝  6*x^4 + 2*x
ejTerm  ⇝  4*x
```

21.1.2. Propiedades del TAD de los polinomios

1. `esPolCero polCero`
2. `n > grado p && b /= 0 ==>`
   `not (esPolCero (consPol n b p))`
3. `consPol (grado p) (coefLider p) (restoPol p) == p`
4. `n > grado p && b /= 0 ==>`
   `grado (consPol n b p) == n`
5. `n > grado p && b /= 0 ==>`
   `coefLider (consPol n b p) == b`
6. `n > grado p && b /= 0 ==>`
   `restoPol (consPol n b p) == p`

21.2. Implementación del TAD de los polinomios

21.2.1. Los polinomios como tipo de dato algebraico

Cabecera del módulo:

```
module PolRepTDA
  ( Polinomio,
    polCero,   -- Polinomio a
    esPolCero, -- Num a =>  Polinomio a -> Bool
    consPol,   -- (Num a) => Int -> a -> Polinomio a
               --              -> Polinomio a
    grado,     -- Polinomio a -> Int
    coefLider, -- Num a => Polinomio a -> a
    restoPol   -- Polinomio a -> Polinomio a
  ) where
```

- Representamos un polinomio mediante los constructores `ConsPol` y `PolCero`.
- Por ejemplo, el polinomio

  ```
  6x^4 -5x^2 + 4x -7
  ```

  se representa por

  ```
  ConsPol 4 6
    (ConsPol 2 (-5)
       (ConsPol 1 4
          (ConsPol 0 (-7) PolCero)))
  ```

- El tipo de los polinomios.

  ```
  data Polinomio a = PolCero
                   | ConsPol Int a (Polinomio a)
                   deriving Eq
  ```

Procedimiento de escritura de los polinomios.

```
instance Num a => Show (Polinomio a) where
    show PolCero               = "0"
    show (ConsPol 0 b PolCero) = show b
    show (ConsPol 0 b p)       = concat [show b," + ",show p]
    show (ConsPol 1 b PolCero) = concat [show b,"*x"]
    show (ConsPol 1 b p)       = concat [show b,"*x + ",show p]
    show (ConsPol n 1 PolCero) = concat ["x^",show n]
    show (ConsPol n b PolCero) = concat [show b,"*x^",show n]
    show (ConsPol n 1 p)       = concat ["x^",show n," + ",show p]
    show (ConsPol n b p)       = concat [show b,"*x^",show n," + ",show p]
```

- `polCero` es el polinomio cero. Por ejemplo,

```
ghci> polCero
0
```

```
polCero :: Polinomio a
polCero = PolCero
```

- `(esPolCero p)` se verifica si `p` es el polinomio cero. Por ejemplo,

```
esPolCero polCero  ~>  True
esPolCero ejPol1   ~>  False
```

```
esPolCero :: Polinomio a -> Bool
esPolCero PolCero = True
esPolCero _       = False
```

- `(consPol n b p)` es el polinomio $bx^n + p$. Por ejemplo,

```
ejPol2                ~>  x^5 + 5*x^2 + 4*x
consPol 3 0 ejPol2    ~>  x^5 + 5*x^2 + 4*x
consPol 3 2 polCero   ~>  2*x^3
consPol 6 7 ejPol2    ~>  7*x^6 + x^5 + 5*x^2 + 4*x
consPol 4 7 ejPol2    ~>  x^5 + 7*x^4 + 5*x^2 + 4*x
consPol 5 7 ejPol2    ~>  8*x^5 + 5*x^2 + 4*x
```

```
consPol :: Num a => Int -> a -> Polinomio a -> Polinomio a
consPol _ 0 p = p
consPol n b PolCero = ConsPol n b PolCero
consPol n b (ConsPol m c p)
    | n > m      = ConsPol n b (ConsPol m c p)
    | n < m      = ConsPol m c (consPol n b p)
    | b+c == 0   = p
    | otherwise  = ConsPol n (b+c) p
```

- `(grado p)` es el grado del polinomio `p`. Por ejemplo,

```
ejPol3        ~>  6*x^4 + 2*x
grado ejPol3  ~>  4
```

```
grado:: Polinomio a -> Int
grado PolCero         = 0
grado (ConsPol n _ _) = n
```

- (`coefLider p`) es el coeficiente líder del polinomio p. Por ejemplo,

```
coefLider ejPol3  ~>  6
```

```
coefLider:: Num t => Polinomio t -> t
coefLider PolCero         = 0
coefLider (ConsPol _ b _) = b
```

- (`restoPol p`) es el resto del polinomio p. Por ejemplo,

```
ejPol3           ~>  6*x^4 + 2*x
restoPol ejPol3  ~>  2*x
ejPol2           ~>  x^5 + 5*x^2 + 4*x
restoPol ejPol2  ~>  5*x^2 + 4*x
```

```
restoPol :: Polinomio t -> Polinomio t
restoPol PolCero         = PolCero
restoPol (ConsPol _ _ p) = p
```

21.2.2. Los polinomios como listas dispersas

Cabecera del módulo

```
module PolRepDispersa
  ( Polinomio,
    polCero,   -- Polinomio a
    esPolCero, -- Num a =>  Polinomio a -> Bool
    consPol,   -- (Num a) => Int -> a -> Polinomio a
               --            -> Polinomio a
    grado,     -- Polinomio a -> Int
    coefLider, -- Num a => Polinomio a -> a
    restoPol   -- Polinomio a -> Polinomio a
  ) where
```

- Representaremos un polinomio por la lista de sus coeficientes ordenados en orden decreciente según el grado.

- Por ejemplo, el polinomio

```
6x^4 -5x^2 + 4x -7
```

  se representa por la lista

```
[6,0,-2,4,-7]
```

- Los polinomios como listas dispersas.

```
data Polinomio a = Pol [a]
                   deriving Eq
```

Procedimiento de escritura de los polinomios.

```
instance Num t => Show (Polinomio t) where
  show pol
      | esPolCero pol         = "0"
      | n == 0 && esPolCero p = show a
      | n == 0                = concat [show a," + ",show p]
      | n == 1 && esPolCero p = concat [show a,"*x"]
      | n == 1                = concat [show a,"*x + ",show p]
      | a == 1 && esPolCero p = concat ["x^",show n]
      | esPolCero p           = concat [show a,"*x^",show n]
      | a == 1                = concat ["x^",show n," + ",show p]
      | otherwise             = concat [show a,"*x^",show n," + ",show p]
     where n = grado pol
           a = coefLider pol
           p = restoPol pol
```

- `polCero` es el polinomio cero. Por ejemplo,

```
ghci> polCero
0
```

```
polCero :: Polinomio a
polCero = Pol []
```

- `(esPolCero p)` se verifica si `p` es el polinomio cero. Por ejemplo,

```
esPolCero polCero  ⇝  True
esPolCero ejPol1   ⇝  False
```

```
esPolCero :: Polinomio a -> Bool
esPolCero (Pol []) = True
esPolCero _        = False
```

- (consPol n b p) es el polinomio $bx^n + p$. Por ejemplo,

```
ejPol2               ⇝  x^5 + 5*x^2 + 4*x
consPol 3 0 ejPol2   ⇝  x^5 + 5*x^2 + 4*x
consPol 3 2 polCero  ⇝  2*x^3
consPol 6 7 ejPol2   ⇝  7*x^6 + x^5 + 5*x^2 + 4*x
consPol 4 7 ejPol2   ⇝  x^5 + 7*x^4 + 5*x^2 + 4*x
consPol 5 7 ejPol2   ⇝  8*x^5 + 5*x^2 + 4*x
```

```
consPol :: Num a => Int -> a -> Polinomio a -> Polinomio a
consPol _ 0 p = p
consPol n b p@(Pol xs)
    | esPolCero p = Pol (b:replicate n 0)
    | n > m       = Pol (b:(replicate (n-m-1) 0)++xs)
    | n < m       = consPol m c (consPol n b (restoPol p))
    | b+c == 0    = Pol (dropWhile (==0) (tail xs))
    | otherwise   = Pol ((b+c):tail xs)
    where
      c = coefLider p
      m = grado p
```

- (grado p) es el grado del polinomio p. Por ejemplo,

```
ejPol3        ⇝  6*x^4 + 2*x
grado ejPol3  ⇝  4
```

```
grado:: Polinomio a -> Int
grado (Pol []) = 0
grado (Pol xs) = length xs - 1
```

- (coefLider p) es el coeficiente líder del polinomio p. Por ejemplo,

```
coefLider ejPol3  ⇝  6
```

```
coefLider:: Num t => Polinomio t -> t
coefLider (Pol [])    = 0
coefLider (Pol (a:_)) = a
```

- `(restoPol p)` es el resto del polinomio `p`. Por ejemplo,

```
ejPol3            ~>  6*x^4 + 2*x
restoPol ejPol3   ~>  2*x
ejPol2            ~>  x^5 + 5*x^2 + 4*x
restoPol ejPol2   ~>  5*x^2 + 4*x
```

```
restoPol :: Num t => Polinomio t -> Polinomio t
restoPol (Pol [])     = polCero
restoPol (Pol [_])    = polCero
restoPol (Pol (_:b:as))
    | b == 0    = Pol (dropWhile (==0) as)
    | otherwise = Pol (b:as)
```

21.2.3. Los polinomios como listas densas

Cabecera del módulo.

```
module PolRepDensa
  ( Polinomio,
    polCero,   -- Polinomio a
    esPolCero, -- Num a =>  Polinomio a -> Bool
    consPol,   -- Num a => Int -> a -> Polinomio a
               --           -> Polinomio a
    grado,     -- Polinomio a -> Int
    coefLider, -- Num a => Polinomio a -> a
    restoPol   -- Polinomio a -> Polinomio a
  ) where
```

- Representaremos un polinomio mediante una lista de pares (grado,coef), ordenados en orden decreciente según el grado. Por ejemplo, el polinomio

```
6x^4 -5x^2 + 4x -7
```

se representa por la lista de pares

```
[(4,6),(2,-5),(1,4),(0,-7)].
```

- Los polinomios como listas densas.

```
data Polinomio a = Pol [(Int,a)]
                   deriving Eq
```

Procedimiento de escritura de polinomios

```
instance Num t => Show (Polinomio t) where
  show pol
      | esPolCero pol         = "0"
      | n == 0 && esPolCero p = show a
      | n == 0                = concat [show a," + ",show p]
      | n == 1 && esPolCero p = concat [show a,"*x"]
      | n == 1                = concat [show a,"*x + ",show p]
      | a == 1 && esPolCero p = concat ["x^",show n]
      | esPolCero p           = concat [show a,"*x^",show n]
      | a == 1                = concat ["x^",show n," + ",show p]
      | otherwise             = concat [show a,"*x^",show n," + ",show p]
     where n = grado pol
           a = coefLider pol
           p = restoPol pol
```

- `polCero` es el polinomio cero. Por ejemplo,

```
ghci> polCero
0
```

```
polCero :: Num a => Polinomio a
polCero = Pol []
```

- `(esPolCero p)` se verifica si p es el polinomio cero. Por ejemplo,

```
esPolCero polCero  ⇝  True
esPolCero ejPol1   ⇝  False
```

```
esPolCero :: Num a => Polinomio a -> Bool
esPolCero (Pol []) = True
esPolCero _        = False
```

- `(consPol n b p)` es el polinomio $bx^n + p$. Por ejemplo,

```
ejPol2                 ⇝  x^5 + 5*x^2 + 4*x
consPol 3 0 ejPol2     ⇝  x^5 + 5*x^2 + 4*x
consPol 3 2 polCero    ⇝  2*x^3
consPol 6 7 ejPol2     ⇝  7*x^6 + x^5 + 5*x^2 + 4*x
consPol 4 7 ejPol2     ⇝  x^5 + 7*x^4 + 5*x^2 + 4*x
consPol 5 7 ejPol2     ⇝  8*x^5 + 5*x^2 + 4*x
```

```
consPol :: Num a => Int -> a -> Polinomio a -> Polinomio a
consPol _ 0 p = p
consPol n b p@(Pol xs)
    | esPolCero p = Pol [(n,b)]
    | n > m       = Pol ((n,b):xs)
    | n < m       = consPol m c (consPol n b (Pol (tail xs)))
    | b+c == 0    = Pol (tail xs)
    | otherwise   = Pol ((n,b+c):(tail xs))
    where
      c = coefLider p
      m = grado p
```

- (`grado p`) es el grado del polinomio p. Por ejemplo,

```
ejPol3        ⇝  6*x^4 + 2*x
grado ejPol3  ⇝  4
```

```
grado:: Polinomio a -> Int
grado (Pol [])         = 0
grado (Pol ((n,_):_)) = n
```

- (`coefLider p`) es el coeficiente líder del polinomio p. Por ejemplo,

```
coefLider ejPol3  ⇝  6
```

```
coefLider:: Num t => Polinomio t -> t
coefLider (Pol [])         = 0
coefLider (Pol ((_,b):_)) = b
```

- (`restoPol p`) es el resto del polinomio p. Por ejemplo,

```
ejPol3            ⇝  6*x^4 + 2*x
restoPol ejPol3   ⇝  2*x
ejPol2            ⇝  x^5 + 5*x^2 + 4*x
restoPol ejPol2   ⇝  5*x^2 + 4*x
```

```
restoPol :: Num t => Polinomio t -> Polinomio t
restoPol (Pol [])     = polCero
restoPol (Pol [_])    = polCero
restoPol (Pol (_:xs)) = Pol xs
```

21.3. Comprobación de las implementaciones con QuickCheck

21.3.1. Librerías auxiliares

- Importación de la implementación a verificar.

```
import PolRepTDA
-- import PolRepDispersa
-- import PolRepDensa
```

- Librerías auxiliares.

```
import Test.QuickCheck
import Test.Framework
import Test.Framework.Providers.QuickCheck2
```

21.3.2. Generador de polinomios

- (genPol n) es un generador de polinomios. Por ejemplo,

```
ghci> sample (genPol 1)
7*x^9 + 9*x^8 + 10*x^7 + -14*x^5 + -15*x^2 + -10
-4*x^8 + 2*x
```

```
genPol :: Int -> Gen (Polinomio Int)
genPol 0 = return polCero
genPol n = do n <- choose (0,10)
              b <- choose (-10,10)
              p <- genPol (div n 2)
              return (consPol n b p)

instance Arbitrary (Polinomio Int) where
    arbitrary = sized genPol
```

21.3.3. Especificación de las propiedades de los polinomios

- `polCero` es el polinomio cero.

```
prop_polCero_es_cero :: Bool
prop_polCero_es_cero =
    esPolCero polCero
```

- Si `n` es mayor que el grado de `p` y `b` no es cero, entonces `(consPol n b p)` es un polinomio distinto del cero.

```
prop_consPol_no_cero :: Int -> Int -> Polinomio Int
                        -> Property
prop_consPol_no_cero n b p =
    n > grado p && b /= 0  ==>
      not (esPolCero (consPol n b p))
```

- `(consPol (grado p) (coefLider p) (restoPol p))` es igual a `p`.

```
prop_consPol :: Polinomio Int -> Bool
prop_consPol p =
    consPol (grado p) (coefLider p) (restoPol p) == p
```

- Si `n` es mayor que el grado de `p` y `b` no es cero, entonces el grado de `(consPol n b p)` es `n`.

```
prop_grado :: Int -> Int -> Polinomio Int -> Property
prop_grado n b p =
    n > grado p && b /= 0 ==>
      grado (consPol n b p) == n
```

- Si `n` es mayor que el grado de `p` y `b` no es cero, entonces el coeficiente líder de `(consPol n b p)` es `b`.

```
prop_coefLider :: Int -> Int -> Polinomio Int -> Property
prop_coefLider n b p =
    n > grado p && b /= 0 ==>
      coefLider (consPol n b p) == b
```

- Si `n` es mayor que el grado de `p` y `b` no es cero, entonces el resto de `(consPol n b p)` es `p`.

```
prop_restoPol :: Int -> Int -> Polinomio Int -> Property
prop_restoPol n b p =
    n > grado p && b /= 0 ==>
      restoPol (consPol n b p) == p
```

21.3.4. Comprobación de las propiedades

Procedimiento de comprobación

- `compruebaPropiedades` comprueba todas las propiedades con la plataforma de verificación. Por ejemplo,

```
compruebaPropiedades =
    defaultMain
        [testGroup "Propiedades del TAD polinomio:"
          [testProperty "P1" prop_polCero_es_cero,
           testProperty "P2" prop_consPol_no_cero,
           testProperty "P3" prop_consPol,
           testProperty "P4" prop_grado,
           testProperty "P5" prop_coefLider,
           testProperty "P6" prop_restoPol]]
```

Comprobación de las propiedades de los polinomios

```
ghci> compruebaPropiedades
Propiedades del TAD polinomio::
  P1: [OK, passed 100 tests]
  P2: [OK, passed 100 tests]
  P3: [OK, passed 100 tests]
  P4: [OK, passed 100 tests]
  P5: [OK, passed 100 tests]
  P6: [OK, passed 100 tests]

         Properties  Total
 Passed  6           6
 Failed  0           0
 Total   6           6
```

21.4. Operaciones con polinomios

21.4.1. Operaciones con polinomios

- Importación de la implementación a utilizar.

```
import PolRepTDA
-- import PolRepDispersa
-- import PolRepDensa
```

- Importación de librerías auxiliares.

```
import Test.QuickCheck
import Test.Framework
import Test.Framework.Providers.QuickCheck2
```

Funciones sobre términos

- `(creaTermino n a)` es el término ax^n. Por ejemplo,

```
creaTermino 2 5  ⇝  5*x^2
```

```
creaTermino:: Num t => Int -> t -> Polinomio t
creaTermino n a = consPol n a polCero
```

- `(termLider p)` es el término líder del polinomio p. Por ejemplo,

```
ejPol2           ⇝  x^5 + 5*x^2 + 4*x
termLider ejPol2  ⇝  x^5
```

```
termLider:: Num t => Polinomio t -> Polinomio t
termLider p = creaTermino (grado p) (coefLider p)
```

Suma de polinomios

- `(sumaPol p q)` es la suma de los polinomios p y q. Por ejemplo,

```
ejPol1                ⇝  3*x^4 + -5*x^2 + 3
ejPol2                ⇝  x^5 + 5*x^2 + 4*x
sumaPol ejPol1 ejPol2  ⇝  x^5 + 3*x^4 + 4*x + 3
```

```
sumaPol:: Num a => Polinomio a -> Polinomio a -> Polinomio a
sumaPol p q
    | esPolCero p = q
    | esPolCero q = p
    | n1 > n2      = consPol n1 a1 (sumaPol r1 q)
    | n1 < n2      = consPol n2 a2 (sumaPol p r2)
    | a1+a2 /= 0   = consPol n1 (a1+a2) (sumaPol r1 r2)
    | otherwise    = sumaPol r1 r2
    where n1 = grado p
          a1 = coefLider p
          r1 = restoPol p
          n2 = grado q
          a2 = coefLider q
          r2 = restoPol q
```

Propiedades de la suma de polinomios

- El polinomio cero es el elemento neutro de la suma.

```
prop_neutroSumaPol :: Polinomio Int -> Bool
prop_neutroSumaPol p =
    sumaPol polCero p == p
```

- La suma es conmutativa.

```
prop_conmutativaSuma :: Polinomio Int -> Polinomio Int
                         -> Bool
prop_conmutativaSuma p q =
    sumaPol p q == sumaPol q p
```

Producto de polinomios

- (multPorTerm t p) es el producto del término t por el polinomio p. Por ejemplo,

```
ejTerm                    ~> 4*x
ejPol2                    ~> x^5 + 5*x^2 + 4*x
multPorTerm ejTerm ejPol2 ~> 4*x^6 + 20*x^3 + 16*x^2
```

```
multPorTerm :: Num t => Polinomio t -> Polinomio t -> Polinomio t
multPorTerm term pol
    | esPolCero pol = polCero
    | otherwise     = consPol (n+m) (a*b) (multPorTerm term r)
    where n = grado term
          a = coefLider term
          m = grado pol
          b = coefLider pol
          r = restoPol pol
```

- (multPol p q) es el producto de los polinomios p y q. Por ejemplo,

```
ghci> ejPol1
3*x^4 + -5*x^2 + 3
ghci> ejPol2
x^5 + 5*x^2 + 4*x
ghci> multPol ejPol1 ejPol2
3*x^9 + -5*x^7 + 15*x^6 + 15*x^5 + -25*x^4 + -20*x^3
      + 15*x^2 + 12*x
```

```
multPol :: Num a => Polinomio a -> Polinomio a -> Polinomio a
multPol p q
    | esPolCero p = polCero
    | otherwise   = sumaPol (multPorTerm (termLider p) q)
                            (multPol (restoPol p) q)
```

Propiedades del producto polinomios

- El producto de polinomios es conmutativo.

```
prop_conmutativaProducto :: Polinomio Int
                            -> Polinomio Int -> Bool
prop_conmutativaProducto p q =
    multPol p q == multPol q p
```

- El producto es distributivo respecto de la suma.

```
prop_distributiva :: Polinomio Int -> Polinomio Int
                     -> Polinomio Int -> Bool
prop_distributiva p q r =
```

```
    multPol p (sumaPol q r) ==
    sumaPol (multPol p q) (multPol p r)
```

Polinomio unidad

- `polUnidad` es el polinomio unidad. Por ejemplo,

```
ghci> polUnidad
1
```

```
polUnidad:: Num t => Polinomio t
polUnidad = consPol 0 1 polCero
```

- El polinomio unidad es el elemento neutro del producto.

```
prop_polUnidad :: Polinomio Int -> Bool
prop_polUnidad p =
    multPol p polUnidad == p
```

Valor de un polinomio en un punto

- `(valor p c)` es el valor del polinomio `p` al sustituir su variable por `c`. Por ejemplo,

```
ejPol1             ~>  3*x^4 + -5*x^2 + 3
valor ejPol1 0     ~>  3
valor ejPol1 1     ~>  1
valor ejPol1 (-2)  ~>  31
```

```
valor:: Num a => Polinomio a -> a -> a
valor p c
    | esPolCero p = 0
    | otherwise   =  b*c^n + valor r c
    where n = grado p
          b = coefLider p
          r = restoPol p
```

Verificación de raices de polinomios

- `(esRaiz c p)` se verifica si `c` es una raiz del polinomio `p`. por ejemplo,

```
ejPol3           ~>  6*x^4 + 2*x
esRaiz 1 ejPol3  ~>  False
esRaiz 0 ejPol3  ~>  True
```

```
esRaiz:: Num a => a -> Polinomio a -> Bool
esRaiz c p = valor p c == 0
```

Derivación de polinomios

- `(derivada p)` es la derivada del polinomio `p`. Por ejemplo,

```
ejPol2           ~>  x^5 + 5*x^2 + 4*x
derivada ejPol2  ~>  5*x^4 + 10*x + 4
```

```
derivada :: Polinomio Int -> Polinomio Int
derivada p
    | n == 0     = polCero
    | otherwise  = consPol (n-1) (n*b) (derivada r)
    where n = grado p
          b = coefLider p
          r = restoPol p
```

Propiedades de las derivadas de polinomios

- La derivada de la suma es la suma de las derivadas.

```
prop_derivada :: Polinomio Int -> Polinomio Int -> Bool
prop_derivada p q =
    derivada (sumaPol p q) ==
    sumaPol (derivada p) (derivada q)
```

Tema 22

Algoritmos sobre grafos

22.1. El TAD de los grafos

22.1.1. Definiciones y terminología sobre grafos

- Un **grafo G** es un par (V, A) donde V es el conjunto de los **vértices** (o nodos) y A el de las **aristas**.
- Una **arista** del grafo es un par de vértices.
- Un **arco** es una arista dirigida.
- **|V|** es el número de vértices.
- **|A|** es el número de aristas.
- Un vértice v es **adjacente** a v' si vv' es una arista del grafo.
- Un **grafo ponderado** es un grafo cuyas aristas tienen un peso.

22.1.2. Signatura del TAD de los grafos

Signatura del TAD de los grafos

```
creaGrafo   :: (Ix v,Num p) => Orientacion -> (v,v) -> [(v,v,p)] ->
                               Grafo v p
dirigido    :: (Ix v,Num p) => (Grafo v p) -> Bool
adyacentes  :: (Ix v,Num p) => (Grafo v p) -> v -> [v]
nodos       :: (Ix v,Num p) => (Grafo v p) -> [v]
aristas     :: (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
aristaEn    :: (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool
peso        :: (Ix v,Num p) => v -> v -> (Grafo v p) -> p
```

Descripción de la signatura del TAD de grafos

- `(creaGrafo d cs as)` es un grafo (dirigido o no, según el valor de o), con el par de cotas cs y listas de aristas as (cada arista es un trío formado por los dos vértices y su peso).
  Ver un ejemplo en la siguiente transparencia.
- `(dirigido g)` se verifica si `g` es dirigido.
- `(nodos g)` es la lista de todos los nodos del grafo `g`.
- `(aristas g)` es la lista de las aristas del grafo `g`.
- `(adyacentes g v)` es la lista de los vértices adyacentes al nodo `v` en el grafo `g`.
- `(aristaEn g a)` se verifica si `a` es una arista del grafo `g`.
- `(peso v1 v2 g)` es el peso de la arista que une los vértices `v1` y `v2` en el grafo `g`.

Ejemplo de creación de grafos.

```
creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
                    (2,4,55),(2,5,32),
                    (3,4,61),(3,5,44),
                    (4,5,93)]
```

crea el grafo

```
      12
  1 -------- 2
  | \78     /|
  |  \   32/ |
  |   \   /  |
34|     5    |55
  |   /   \  |
  |  /44   \ |
  | /     93\|
  3 -------- 4
      61
```

22.1.3. Implementación de los grafos como vectores de adyacencia

- Cabecera del módulo:

```
module GrafoConVectorDeAdyacencia
    (Orientacion (..),
     Grafo,
     creaGrafo,  -- (Ix v,Num p) => Orientacion -> (v,v) -> [(v,v,p)] ->
                 --                 Grafo v p
     dirigido,   -- (Ix v,Num p) => (Grafo v p) -> Bool
     adyacentes, -- (Ix v,Num p) => (Grafo v p) -> v -> [v]
     nodos,      -- (Ix v,Num p) => (Grafo v p) -> [v]
     aristas,    -- (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
     aristaEn,   -- (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool
     peso        -- (Ix v,Num p) => v -> v -> (Grafo v p) -> p
    ) where
```

- Librerías auxiliares.

```
import Data.Array
```

- `Orientacion` es `D` (dirigida) ó `ND` (no dirigida).

```
data Orientacion = D | ND
                   deriving (Eq, Show)
```

- `(Grafo v p)` es un grafo con vértices de tipo `v` y pesos de tipo `p`.

```
data Grafo v p = G Orientacion (Array v [(v,p)])
                 deriving (Eq, Show)
```

- `(creaGrafo o cs as)` es un grafo (dirigido o no según el valor de `o`), con el par de cotas `cs` y listas de aristas `as` (cada arista es un trío formado por los dos vértices y su peso). Ver un ejemplo a continuación.

```
creaGrafo :: (Ix v, Num p) =>
             Orientacion -> (v,v) -> [(v,v,p)] -> Grafo v p
creaGrafo o cs vs =
    G o (accumArray
         (\xs x -> xs++[x]) [] cs
         ((if o == D then []
```

```
           else [(x2,(x1,p))|(x1,x2,p) <- vs, x1 /= x2]) ++
          [(x1,(x2,p)) | (x1,x2,p) <- vs]))
```

- `ejGrafoND` es el grafo que de la página **??**. Por ejemplo,

```
ghci> ejGrafoND
G ND array (1,5) [(1,[(2,12),(3,34),(5,78)]),
                  (2,[(1,12),(4,55),(5,32)]),
                  (3,[(1,34),(4,61),(5,44)]),
                  (4,[(2,55),(3,61),(5,93)]),
                  (5,[(1,78),(2,32),(3,44),(4,93)])])
```

```
ejGrafoND = creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
                                (2,4,55),(2,5,32),
                                (3,4,61),(3,5,44),
                                (4,5,93)]
```

- `ejGrafoD` es el mismo grafo que ejGrafoND pero orientando las aristas de menor a mayor. Por ejemplo,

```
ghci> ejGrafoD
G D array (1,5) [(1,[(2,12),(3,34),(5,78)]),
                 (2,[(4,55),(5,32)]),
                 (3,[(4,61),(5,44)]),
                 (4,[(5,93)]),
                 (5,[])])
```

```
ejGrafoD = creaGrafo D (1,5) [(1,2,12),(1,3,34),(1,5,78),
                              (2,4,55),(2,5,32),
                              (3,4,61),(3,5,44),
                              (4,5,93)]
```

- `(dirigido g)` se verifica si `g` es dirigido. Por ejemplo,

```
dirigido ejGrafoD   ==  True
dirigido ejGrafoND  ==  False
```

```
dirigido :: (Ix v,Num p) => (Grafo v p) -> Bool
dirigido (G o _) = o == D
```

- (adyacentes g v) es la lista de los vértices adyacentes al nodo v en el grafo g. Por ejemplo,

```
adyacentes ejGrafoND 4  ==  [2,3,5]
adyacentes ejGrafoD  4  ==  [5]
```

```
adyacentes :: (Ix v,Num p) => (Grafo v p) -> v -> [v]
adyacentes (G _ g) v = map fst (g!v)
```

- (nodos g) es la lista de todos los nodos del grafo g. Por ejemplo,

```
nodos ejGrafoND  ==  [1,2,3,4,5]
nodos ejGrafoD   ==  [1,2,3,4,5]
```

```
nodos :: (Ix v,Num p) => (Grafo v p) -> [v]
nodos (G _ g) = indices g
```

- (peso v1 v2 g) es el peso de la arista que une los vértices v1 y v2 en el grafo g. Por ejemplo,

```
peso 1 5 ejGrafoND  ==  78
peso 1 5 ejGrafoD   ==  78
```

```
peso :: (Ix v,Num p) => v -> v -> (Grafo v p) -> p
peso x y (G _ g) = head [c | (a,c) <- g!x , a == y]
```

- (aristaEn g a) se verifica si a es una arista del grafo g. Por ejemplo,

```
aristaEn ejGrafoND (5,1)  ==  True
aristaEn ejGrafoND (4,1)  ==  False
aristaEn ejGrafoD  (5,1)  ==  False
aristaEn ejGrafoD  (1,5)  ==  True
```

```
aristaEn :: (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool
aristaEn g (x,y) = y `elem` adyacentes g x
```

- (aristas g) es la lista de las aristas del grafo g. Por ejemplo,

```
ghci> aristas ejGrafoND
[(1,2,12),(1,3,34),(1,5,78),(2,1,12),(2,4,55),(2,5,32),
 (3,1,34),(3,4,61),(3,5,44),(4,2,55),(4,3,61),(4,5,93),
 (5,1,78),(5,2,32),(5,3,44),(5,4,93)]
ghci> aristas ejGrafoD
[(1,2,12),(1,3,34),(1,5,78),(2,4,55),(2,5,32),(3,4,61),
 (3,5,44),(4,5,93)]
```

```
aristas :: (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
aristas (G o g) =
    [(v1,v2,w) | v1 <- nodos (G o g) , (v2,w) <- g!v1]
```

22.1.4. Implementación de los grafos como matrices de adyacencia

- Cabecera del módulo.

```
module GrafoConMatrizDeAdyacencia
    (Orientacion (..),
     Grafo,
     creaGrafo,  -- (Ix v,Num p) => Orientacion -> (v,v) -> [(v,v,p)] ->
                 --                 Grafo v p
     dirigido,   -- (Ix v,Num p) => (Grafo v p) -> Bool
     adyacentes, -- (Ix v,Num p) => (Grafo v p) -> v -> [v]
     nodos,      -- (Ix v,Num p) => (Grafo v p) -> [v]
     aristas,    -- (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
     aristaEn,   -- (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool
     peso        -- (Ix v,Num p) => v -> v -> (Grafo v p) -> p
    ) where
```

- Librerías auxiliares

```
import Data.Array
```

- `Orientacion` es `D` (dirigida) ó `ND` (no dirigida).

```
data Orientacion = D | ND
                   deriving (Eq, Show)
```

- `(Grafo v p)` es un grafo con vértices de tipo `v` y pesos de tipo `p`.

```
data Grafo v p = G Orientacion (Array (v,v) (Maybe p))
                 deriving (Eq, Show)
```

- `(creaGrafo o cs as)` es un grafo (dirigido o no, según el valor de `o`), con el par de cotas `cs` y listas de aristas as (cada arista es un trío formado por los dos vértices y su peso). Ver un ejemplo a continuación.

```
creaGrafo :: (Ix v, Num p) => Bool -> (v,v) -> [(v,v,p)]
                             -> Grafo v p
creaGrafo o cs@(l,u) as
    = G o (matrizVacia //
           ([((x1,x2),Just w) | (x1,x2,w) <- as] ++
            if o == D then []
            else [((x2,x1),Just w) | (x1,x2,w) <- as, x1 /= x2]))
      where
      matrizVacia = array ((l,l),(u,u))
                          [((x1,x2),Nothing) | x1 <- range cs,
                                               x2 <- range cs]
```

- `ejGrafoND` es el grafo que de la página **??**. Por ejemplo,

```
ghci> ejGrafoND
G ND array ((1,1),(5,5))
           [((1,1),Nothing),((1,2),Just 12),((1,3),Just 34),
            ((1,4),Nothing),((1,5),Just 78),((2,1),Just 12),
            ((2,2),Nothing),((2,3),Nothing),((2,4),Just 55),
            ((2,5),Just 32),((3,1),Just 34),((3,2),Nothing),
            ((3,3),Nothing),((3,4),Just 61),((3,5),Just 44),
            ((4,1),Nothing),((4,2),Just 55),((4,3),Just 61),
            ((4,4),Nothing),((4,5),Just 93),((5,1),Just 78),
            ((5,2),Just 32),((5,3),Just 44),((5,4),Just 93),
            ((5,5),Nothing)]
```

```
ejGrafoND = creaGrafo ND (1,5) [(1,2,12),(1,3,34),(1,5,78),
                                (2,4,55),(2,5,32),
                                (3,4,61),(3,5,44),
                                (4,5,93)]
```

- `ejGrafoD` es el mismo grafo que ejGrafoND pero orientando las aristas de menor a mayor. Por ejemplo,

```
ghci> ejGrafoD
G D (array ((1,1),(5,5))
           [((1,1),Nothing),((1,2),Just 12),((1,3),Just 34),
            ((1,4),Nothing),((1,5),Just 78),((2,1),Nothing),
            ((2,2),Nothing),((2,3),Nothing),((2,4),Just 55),
            ((2,5),Just 32),((3,1),Nothing),((3,2),Nothing),
            ((3,3),Nothing),((3,4),Just 61),((3,5),Just 44),
            ((4,1),Nothing),((4,2),Nothing),((4,3),Nothing),
            ((4,4),Nothing),((4,5),Just 93),((5,1),Nothing),
            ((5,2),Nothing),((5,3),Nothing),((5,4),Nothing),
            ((5,5),Nothing)])
```

```
ejGrafoD = creaGrafo D (1,5) [(1,2,12),(1,3,34),(1,5,78),
                              (2,4,55),(2,5,32),
                              (3,4,61),(3,5,44),
                              (4,5,93)]
```

- (dirigido g) se verifica si g es dirigido. Por ejemplo,

```
dirigido ejGrafoD   ==  True
dirigido ejGrafoND  ==  False
```

```
dirigido :: (Ix v,Num p) => (Grafo v p) -> Bool
dirigido (G o _) = o == D
```

- (adyacentes g v) es la lista de los vértices adyacentes al nodo v en el grafo g. Por ejemplo,

```
adyacentes ejGrafoND 4  ==  [2,3,5]
adyacentes ejGrafoD  4  ==  [5]
```

```
adyacentes :: (Ix v,Num p) => (Grafo v p) -> v -> [v]
adyacentes (G o g) v =
    [v' | v' <- nodos (G o g), (g!(v,v')) /= Nothing]
```

- (nodos g) es la lista de todos los nodos del grafo g. Por ejemplo,

```
nodos ejGrafoND  ==  [1,2,3,4,5]
nodos ejGrafoD   ==  [1,2,3,4,5]
```

```
nodos :: (Ix v,Num p) => (Grafo v p) -> [v]
nodos (G _ g) = range (l,u)
    where ((l,_),(u,_)) = bounds g
```

- (peso v1 v2 g) es el peso de la arista que une los vértices v1 y v2 en el grafo g. Por ejemplo,

```
peso 1 5 ejGrafoND  ==  78
peso 1 5 ejGrafoD   ==  78
```

```
peso :: (Ix v,Num p) => v -> v -> (Grafo v p) -> p
peso x y (G _ g)  = w where (Just w) = g!(x,y)
```

- (aristaEn g a) se verifica si a es una arista del grafo g. Por ejemplo,

```
aristaEn ejGrafoND (5,1)  ==  True
aristaEn ejGrafoND (4,1)  ==  False
```

```
aristaEn :: (Ix v,Num p) => (Grafo v p) -> (v,v) -> Bool
aristaEn (G _o g) (x,y)= (g!(x,y)) /= Nothing
```

- (aristas g) es la lista de las aristas del grafo g. Por ejemplo,

```
ghci> aristas ejGrafoD
[(1,2,12),(1,3,34),(1,5,78),(2,4,55),(2,5,32),(3,4,61),
 (3,5,44),(4,5,93)]
ghci> aristas ejGrafoND
[(1,2,12),(1,3,34),(1,5,78),(2,1,12),(2,4,55),(2,5,32),
 (3,1,34),(3,4,61),(3,5,44),(4,2,55),(4,3,61),(4,5,93),
 (5,1,78),(5,2,32),(5,3,44),(5,4,93)]
```

```
aristas :: (Ix v,Num p) => (Grafo v p) -> [(v,v,p)]
aristas g@(G o e) = [(v1,v2,extrae(e!(v1,v2)))
                     | v1 <- nodos g,
                       v2 <- nodos g,
                       aristaEn g (v1,v2)]
    where extrae (Just w) = w
```

22.2. Recorridos en profundidad y en anchura

22.2.1. Recorrido en profundidad

- Importaciones de librerías auxiliares.

```
-- Nota: Elegir una implementación de los grafos.
import GrafoConVectorDeAdyacencia
-- import GrafoConMatrizDeAdyacencia
```

- En los ejemplos se usará el grafo g

```
+---> 2 <---+
|           |
|           |
1 --> 3 --> 6 --> 5
|                 |
|                 |
+---> 4 <---------+
```

que se define por

```
g = creaGrafo D (1,6)
              [(1,2,0),(1,3,0),(1,4,0),(3,6,0),
               (5,4,0),(6,2,0),(6,5,0)]
```

Procedimiento elemental de recorrido en profundidad

- (recorridoEnProfundidad i g) es el recorrido en profundidad del grafo g desde el vértice i. Por ejemplo,

```
recorridoEnProfundidad 1 g  ==  [1,2,3,6,5,4]
```

```
recorridoEnProfundidad i g = rp [i] []
    where
      rp [] vis    = vis
      rp (c:cs) vis
          | c `elem` vis = rp cs vis
          | otherwise    = rp ((adyacentes g c)++cs)
                              (vis++[c])
```

- Traza del cálculo de (recorridoEnProfundidad 1 g)

```
recorridoEnProfundidad 1 g
= rp [1]     []
= rp [2,3,4] [1]
= rp [3,4]   [1,2]
= rp [6,4]   [1,2,3]
= rp [2,5,4] [1,2,3,6]
= rp [5,4]   [1,2,3,6]
= rp [4,4]   [1,2,3,6,5]
= rp [4]     [1,2,3,6,5,4]
= rp []      [1,2,3,6,5,4]
= [1,2,3,6,5,4]
```

Recorrido en profundidad con acumuladores

- (`recorridoEnProfundidad' i g`) es el recorrido en profundidad del grafo, usando la lista de los visitados como acumulador. Por ejemplo,

```
recorridoEnProfundidad' 1 g  ==  [1,2,3,6,5,4]
```

```
recorridoEnProfundidad' i g = reverse (rp [i] [])
    where
      rp [] vis     = vis
      rp (c:cs) vis
          | c `elem` vis = rp cs vis
          | otherwise    = rp ((adyacentes g c)++cs)
                              (c:vis)
```

- Traza del cálculo de (`recorridoEnProfundidad' 1 g`)

```
recorridoEnProfundidad' 1 g
= reverse (rp [1]     [])
= reverse (rp [2,3,4] [1])
= reverse (rp [3,4]   [2,1])
= reverse (rp [6,4]   [3,2,1])
= reverse (rp [2,5,4] [6,3,2,1])
= reverse (rp [5,4]   [6,3,2,1])
= reverse (rp [4,4]   [5,6,3,2,1])
= reverse (rp [4]     [4,5,6,3,2,1])
= reverse (rp []      [4,5,6,3,2,1])
= reverse [4,5,6,3,2,1]
= [1,2,3,6,5,4]
```

22.2.2. Recorrido en anchura

- Importaciones de librerías auxiliares.

```
-- Nota: Elegir una implementación de los grafos.
import GrafoConVectorDeAdyacencia
-- import GrafoConMatrizDeAdyacencia
```

Procedimiento elemental de recorrido en anchura

- (`recorridoEnAnchura i g`) es el recorrido en anchura del grafo `g` desde el vértice `i`. Por ejemplo,

```
recorridoEnAnchura 1 g  ==  [1,4,3,2,6,5]
```

```
recorridoEnAnchura i g = reverse (ra [i] [])
    where
      ra [] vis    = vis
      ra (c:cs) vis
          | c `elem` vis = ra cs vis
          | otherwise    = ra (cs ++ adyacentes g c)
                              (c:vis)
```

- Traza del cálculo de (`recorridoEnAnchura 1 g`)

```
RecorridoEnAnchura 1 g
= ra [1]     []
= ra [2,3,4] [1]
= ra [3,4]   [2,1]
= ra [4,6]   [3,2,1]
= ra [6]     [4,3,2,1]
= ra [2,5]   [6,4,3,2,1]
= ra [5]     [6,4,3,2,1]
= ra [4]     [5,6,4,3,2,1]
= ra []      [5,6,4,3,2,1]
= [1,2,3,4,6,5]
```

22.3. Árboles de expansión mínimos

22.3.1. Árboles de expansión mínimos

- Sea $G = (V, A)$ un grafo conexo no orientado en el que cada arista tiene un peso no negativo. Un **árbol de expansión mínimo** de G es un subgrafo $G' = (V, A')$ que conecta todos los vértices de G y tal que la suma de sus pesos es mínima.
- **Aplicación:** Si los vértices representan ciudades y el coste de una arista $\{a, b\}$ es el construir una carretera de a a b, entonces un árbol de expansión mínimo representa el modo de enlazar todas las ciudades mediante una red de carreteras de coste mínimo.
- Terminología de algoritmos voraces: Sea $G = (V, A)$ un grafo y T un conjunto de aristas de G.
  - T es una **solución** si es un grafo de expansión.
  - T es **completable** si no tiene ciclos.
  - T es **prometedor** si es completable y puede ser completado hasta llegar a una solución óptima.
  - Una arista **toca** un conjunto de vértices B si exactamente uno de sus extremos pertenece a B.
- **Teorema:** Sea $G = (V, A)$ un grafo conexo no orientado cuyas aristas tienen un peso asociado. Sea B un subjconjunto propio del conjunto de vértices V y T un conjunto prometedor de aristas tal que ninguna arista de T toca a B. Sea e una arista de peso mínimo de entre todas las que tocan a B. Entonces $(T \cup \{e\})$ es prometedor.

22.3.2. El algoritmo de Kruskal

Para los ejemplos se considera el siguiente grafo:

```
    1       2
 1 ----- 2 ----- 3
 |      /|      /|
 |     / |     / |
 |    /  |    /  |
4|   /6  |4  /5  |6
 |  /    |  /    |
 | /     | /     |
 |/      |/      |
```

```
4 ----- 5 ----- 6
 \  3   |   8  /
  \     |     /
   \    |    /
   4\   |7  /3
     \  |  /
      \ | /
       \|/
        7
```

- Aplicación del algoritmo de Kruskal al grafo anterior:

```
Etapa Arista Componentes conexas
0            {1} {2} {3} {4} {5} {6} {7}
1     {1,2}  {1,2} {3} {4} {5} {6} {7}
2     {2,3}  {1,2,3} {4} {5} {6} {7}
3     {4,5}  {1,2,3} {4,5} {6} {7}
4     {6,7}  {1,2,3} {4,5} {6,7}
5     {1,4}  {1,2,3,4,5} {6,7}
6     {2,5}  arista rechazada
7     {4,7}  {1,2,3,4,5,6,7}
```

- El árbol de expansión mínimo contiene las aristas no rechazadas:

```
{1,2}, {2,3}, {4,5}, {6,7}, {1,4} y {4,7}.
```

- Librerías auxiliares.

```
-- Nota: Seleccionar una implementación del TAD grafo.
import GrafoConVectorDeAdyacencia
-- import GrafoConMatrizDeAdyacencia

-- Nota: Seleccionar una implementación del TAD tabla.
-- import TablaConFunciones
import TablaConListasDeAsociacion
-- import TablaConMatrices

import Data.List
import Data.Ix
```

- Grafos usados en los ejemplos.

```
g1 :: Grafo Int Int
g1 = creaGrafo D (1,5) [(1,2,12),(1,3,34),(1,5,78),
                        (2,4,55),(2,5,32),
                        (3,4,61),(3,5,44),
                        (4,5,93)]

g2 :: Grafo Int Int
g2 = creaGrafo D (1,5) [(1,2,13),(1,3,11),(1,5,78),
                        (2,4,12),(2,5,32),
                        (3,4,14),(3,5,44),
                        (4,5,93)]
```

- (`kruskal g`) es el árbol de expansión mínimo del grafo `g` calculado mediante el algoritmo de Kruskal. Por ejemplo,

```
kruskal g1  ==  [(55,2,4),(34,1,3),(32,2,5),(12,1,2)]
kruskal g2  ==  [(32,2,5),(13,1,2),(12,2,4),(11,1,3)]
```

```
kruskal :: (Ix v, Num p, Ord p) => Grafo v p -> [(p,v,v)]
kruskal g = kruskal' cola                           -- Cola de prioridad
                     (tabla [(x,x) | x <- nodos g]) -- Tabla de raices
                     []                             -- Árbol de expansión
                     ((length (nodos g)) - 1)       -- Aristas por
                                                    -- colocar
    where cola = sort [(p,x,y) | (x,y,p) <- aristas g]

kruskal' ((p,x,y):as) t ae n
    | n==0        = ae
    | actualizado = kruskal' as t' ((p,x,y):ae) (n-1)
    | otherwise   = kruskal' as t  ae           n
    where (actualizado,t') = buscaActualiza (x,y) t
```

- (`raiz t n`) es la raíz de `n` en la tabla `t`. Por ejemplo,

```
> raiz (crea [(1,1),(3,1),(4,3),(5,4),(2,6),(6,6)]) 5
1
> raiz (crea [(1,1),(3,1),(4,3),(5,4),(2,6),(6,6)]) 2
6
```

```
raiz:: Eq n => Tabla n n -> n -> n
raiz t x | v == x    = v
         | otherwise = raiz t v
         where v = valor t x
```

- (buscaActualiza a t) es el par formado por False y la tabla t, si los dos vértices de la arista a tienen la misma raíz en t y el par formado por True y la tabla obtenida añadiéndole a t la arista formada por el vértice de a de mayor raíz y la raíz del vértice de a de menor raíz. Por ejemplo,

```
ghci> let t = crea [(1,1),(2,2),(3,1),(4,1)]
ghci> buscaActualiza (2,3) t
(True,Tbl [(1,1),(2,1),(3,1),(4,1)])
ghci> buscaActualiza (3,4) t
(False,Tbl [(1,1),(2,2),(3,1),(4,1)])
```

```
buscaActualiza :: (Eq n, Ord n) => (n,n) -> Tabla n n
                                  -> (Bool,Tabla n n)
buscaActualiza (x,y) t
    | x' == y'  = (False, t)
    | y' <  x'  = (True, modifica (x,y') t)
    | otherwise = (True, modifica (y,x') t)
    where x' = raiz t x
          y' = raiz t y
```

22.3.3. El algoritmo de Prim

- (prim g) es el árbol de expansión mínimo del grafo g calculado mediante el algoritmo de Prim. Por ejemplo,

```
prim g1  ==  [(55,2,4),(34,1,3),(32,2,5),(12,1,2)]
prim g2  ==  [(32,2,5),(12,2,4),(13,1,2),(11,1,3)]
```

```
prim :: (Ix v, Num p, Ord p) => Grafo v p -> [(p,v,v)]
prim g = prim' [n]             -- Nodos colocados
               ns              -- Nodos por colocar
               []              -- Árbol de expansión
               (aristas g)     -- Aristas del grafo
         where (n:ns) = nodos g
```

```
prim' t [] ae as = ae
prim' t r  ae as = prim' (v':t) (delete v' r) (e:ae) as
    where e@(c,u', v') = minimum [(c,u,v)| (u,v,c) <- as,
                                           elem u t,
                                           elem v r]
```

Tema 23

Técnicas de diseño descendente de algoritmos

23.1. La técnica de divide y vencerás

23.1.1. La técnica de divide y vencerás

La técnica **divide y vencerás** consta de los siguientes pasos:

1. *Dividir* el problema en subproblemas menores.
2. *Resolver* por separado cada uno de los subproblemas:
   - si los subproblemas son complejos, usar la misma técnica recursivamente;
   - si son simples, resolverlos directamente.
3. *Combinar* todas las soluciones de los subproblemas en una solución simple.

- `(divideVenceras ind resuelve divide combina pbInicial)` resuelve el problema `pbInicial` mediante la técnica de divide y vencerás, donde
  - `(ind pb)` se verifica si el problema `pb` es indivisible,
  - `(resuelve pb)` es la solución del problema indivisible `pb`,
  - `(divide pb)` es la lista de subproblemas de `pb`,
  - `(combina pb ss)` es la combinación de las soluciones `ss` de los subproblemas del problema `pb` y
  - `pbInicial` es el problema inicial.

```
divideVenceras :: (p -> Bool) -> (p -> s) -> (p -> [p])
                  -> (p -> [s] -> s) -> p -> s
divideVenceras ind resuelve divide combina pbInicial =
    dv' pbInicial where
    dv' pb
        | ind pb    = resuelve pb
        | otherwise = combina pb [dv' sp | sp <- divide pb]
```

23.1.2. La ordenación por mezcla como ejemplo de DyV

- (ordenaPorMezcla xs) es la lista obtenida ordenando xs por el procedimiento de ordenación por mezcla. Por ejemplo,

```
ghci> ordenaPorMezcla [3,1,4,1,5,9,2,8]
[1,1,2,3,4,5,8,9]
```

```
ordenaPorMezcla :: Ord a => [a] -> [a]
ordenaPorMezcla xs =
    divideVenceras ind id divide combina xs
    where
      ind xs            = length xs <= 1
      divide xs         = [take n xs, drop n xs]
                          where n = length xs `div` 2
      combina _ [l1,l2] = mezcla l1 l2
```

- (mezcla xs ys) es la lista obtenida mezclando xs e ys. Por ejemplo,

```
mezcla [1,3] [2,4,6]  ~>  [1,2,3,4,6]
```

```
mezcla :: Ord a => [a] -> [a] -> [a]
mezcla [] b = b
mezcla a [] = a
mezcla a@(x:xs) b@(y:ys)
    | x <= y    = x : (mezcla xs b)
    | otherwise = y : (mezcla a ys)
```

23.1.3. La ordenación rápida como ejemplo de DyV

- (ordenaRapida xs) es la lista obtenida ordenando xs por el procedimiento de ordenación rápida. Por ejemplo,

```
ghci> ordenaRapida [3,1,4,1,5,9,2,8]
[1,1,2,3,4,5,8,9]
```

```
ordenaRapida :: Ord a => [a] -> [a]
ordenaRapida xs =
    divideVenceras ind id divide combina xs
    where
      ind xs               = length xs <= 1
      divide (x:xs)        = [[ y | y <- xs, y <= x],
                               [ y | y <- xs, y > x]]
      combina (x:_) [l1,l2] = l1 ++ [x] ++ l2
```

23.2. Búsqueda en espacios de estados

23.2.1. El patrón de búsqueda en espacios de estados

Descripción de los problemas de espacios de estados

Las características de los problemas de espacios de estados son:

- un conjunto de las posibles situaciones o **nodos** que constituye el **espacio de estados** (estos son las potenciales soluciones que se necesitan explorar),
- un conjunto de movimientos de un nodo a otros nodos, llamados los **sucesores** del nodo,
- un **nodo inicial** y
- un **nodo objetivo** que es la solución.
- En estos problemas usaremos las siguientes librerías auxiliares:

```
-- Nota: Hay que elegir una implementación de las pilas.
import PilaConListas
-- import PilaConTipoDeDatoAlgebraico

import Data.Array
import Data.List (sort)
```

El patrón de búsqueda en espacios de estados

- Se supone que el grafo implícito de espacios de estados es acíclico.
- (buscaEE s o e) es la lista de soluciones del problema de espacio de estado definido por la función sucesores (s), el objetivo (o) y estado inicial (e).

```
buscaEE:: (Eq node) => (node -> [node])  -> (node -> Bool)
                       -> node -> [node]
buscaEE sucesores esFinal x = busca' (apila x vacia)
 where busca' p
    | esVacia p        = []
    | esFinal (cima p) = cima p : busca' (desapila p)
    | otherwise        = busca' (foldr apila (desapila p)
                                       (sucesores x))
                         where x = cima p
```

23.2.2. El problema del 8 puzzle

El problema del 8 puzzle

Para el 8–puzzle se usa un cajón cuadrado en el que hay situados 8 bloques cuadrados. El cuadrado restante está sin rellenar. Cada bloque tiene un número. Un bloque adyacente al hueco puede deslizarse hacia él. El juego consiste en transformar la posición inicial en la posición final mediante el deslizamiento de los bloques. En particular, consideramos el estado inicial y final siguientes:

```
+---+---+---+                   +---+---+---+
| 2 | 6 | 3 |                   | 1 | 2 | 3 |
+---+---+---+                   +---+---+---+
| 5 |   | 4 |                   | 8 |   | 4 |
+---+---+---+                   +---+---+---+
| 1 | 7 | 8 |                   | 7 | 6 | 5 |
+---+---+---+                   +---+---+---+
Estado inicial                  Estado final
```

- Una posición es un par de enteros.

```
type Posicion = (Int,Int)
```

- Un tablero es un vector de posiciones, en el que el índice indica el elemento que ocupa la posición.

```
type Tablero  = Array Int Posicion
```

- `inicial8P` es el estado inicial del 8 puzzle. En el ejemplo es

```
+---+---+---+
| 2 | 6 | 3 |
+---+---+---+
| 5 |   | 4 |
+---+---+---+
| 1 | 7 | 8 |
+---+---+---+
```

```
inicial8P :: Tablero
inicial8P = array (0,8) [(2,(1,3)),(6,(2,3)),(3,(3,3)),
                         (5,(1,2)),(0,(2,2)),(4,(3,2)),
                         (1,(1,1)),(7,(2,1)),(8,(3,1))]
```

- `final8P` es el estado final del 8 puzzle. En el ejemplo es

```
+---+---+---+
| 1 | 2 | 3 |
+---+---+---+
| 8 |   | 4 |
+---+---+---+
| 7 | 6 | 5 |
+---+---+---+
```

```
final8P :: Tablero
final8P = array (0,8) [(1,(1,3)),(2,(2,3)),(3,(3,3)),
                       (8,(1,2)),(0,(2,2)),(4,(3,2)),
                       (7,(1,1)),(6,(2,1)),(5,(3,1))]
```

- `(distancia p1 p2)` es la distancia Manhatan entre las posiciones `p1` y `p2`. Por ejemplo,

```
distancia (2,7) (4,1)  ⇝  8
```

```
distancia :: Posicion -> Posicion -> Int
distancia (x1,y1) (x2,y2) = abs (x1-x2) + abs (y1-y2)
```

- (adyacente p1 p2) se verifica si las posiciones p1 y p2 son adyacentes. Por ejemplo,

```
adyacente (3,2) (3,1)  ~>  True
adyacente (3,2) (1,2)  ~>  False
```

```
adyacente :: Posicion -> Posicion -> Bool
adyacente p1 p2 = distancia p1 p2 == 1
```

- (todosMovimientos t) es la lista de los tableros obtenidos aplicándole al tablero t todos los posibles movimientos; es decir, intercambiando la posición del hueco con sus adyacentes.

```
todosMovimientos :: Tablero -> [Tablero]
todosMovimientos t =
    [t//[(0,t!i),(i,t!0)] | i<-[1..8],
                            adyacente (t!0) (t!i)]
```

- Los nodos del espacio de estados son listas de tableros $[t_n, \ldots, t_1]$ tal que t_i es un sucesor de t_{i-1}.

```
data Tableros = Est [Tablero] deriving (Eq, Show)
```

- (sucesores8P e) es la lista de sucesores del estado e.

```
sucesores8P :: Tableros -> [Tableros]
sucesores8P (Est(n@(t:ts))) =
    filter (noEn ts)
           [Est (t':n) | t' <- todosMovimientos t]
    where
      noEn ts (Est(t:_)) =
         not (elem (elems t) (map elems ts))
```

Solución del 8 puzzle por búsqueda en espacios de estados

- (esFinal8P e) se verifica si e es un estado final del 8 puzzle.

```
esFinal8P :: Tableros -> Bool
esFinal8P (Est (n:_)) = elems n == elems final8P
```

- (buscaEE8P) es la lista de las soluciones del problema del 8 puzzle.

```
buscaEE8P :: [[Posicion]]
buscaEE8P = map elems ls
    where ((Est ls):_) = buscaEE sucesores8P
                                esFinal8P
                                (Est [inicial8P])
```

- Nota: No termina.

23.2.3. El problema de las n reinas

El problema de las n reinas

- El problema de las n reinas consiste en colocar n reinas en un tablero cuadrado de dimensiones n por n de forma que no se encuentren más de una en la misma línea: horizontal, vertical o diagonal.
- Las posiciones de las reinas en el tablero se representan por su columna y su fila.

```
type Columna = Int
type Fila    = Int
```

- Una solución del problema de las n reinas es una lista de posiciones.

```
type SolNR = [(Columna,Fila)]
```

- `(valida sp p)` se verifica si la posición `p` es válida respecto de la solución parcial `sp`; es decir, la reina en la posición `p` no amenaza a ninguna de las reinas de la `sp` (se supone que están en distintas columnas). Por ejemplo,

```
valida [(1,1)] (2,2)  ⇝  False
valida [(1,1)] (2,3)  ⇝  True
```

```
valida :: SolNR -> (Columna,Fila) -> Bool
valida solp (c,r) = and [test s | s <- solp]
    where test (c',r') = and [c'+r'/=c+r,
                              c'-r'/=c-r,
                              r'/=r]
```

- Los nodos del problema de las n reinas son ternas formadas por la columna de la última reina colocada, el número de columnas del tablero y la solución parcial de las reinas colocadas anteriormente.

```
type NodoNR = (Columna,Columna,SolNR)
```

- (`sucesoresNR e`) es la lista de los sucesores del estado e en el problema de las n reinas. Por ejemplo,

```
ghci> sucesoresNR (1,4,[])
[(2,4,[(1,1)]),(2,4,[(1,2)]),(2,4,[(1,3)]),(2,4,[(1,4)])]
```

```
sucesoresNR :: NodoNR -> [NodoNR]
sucesoresNR (c,n,solp)
    = [(c+1,n,solp++[(c,r)]) | r <- [1..n],
                               valida solp (c,r)]
```

- (`esFinalNQ e`) se verifica si e es un estado final del problema de las n reinas.

```
esFinalNQ :: NodoNR -> Bool
esFinalNQ (c,n,solp) = c > n
```

Solución del problema de las n reinas por EE

- (`buscaEE_NQ n`) es la primera solución del problema de las n reinas, por búsqueda en espacio de estados. Por ejemplo,

```
ghci> buscaEE_NQ 8
[(1,1),(2,5),(3,8),(4,6),(5,3),(6,7),(7,2),(8,4)]
```

```
buscaEE_NQ :: Columna -> SolNR
buscaEE_NQ n = s
    where ((_,_,s):_) = buscaEE sucesoresNR
                                esFinalNQ
                                (1,n,[])
```

- (`nSolucionesNQ n`) es el número de soluciones del problema de las n reinas, por búsqueda en espacio de estados. Por ejemplo,

```
nSolucionesNQ 8  ⇝  92
```

```
nSolucionesNQ :: Columna -> Int
nSolucionesNQ n =
    length (buscaEE sucesoresNR
                    esFinalNQ
                    (1,n,[]))
```

23.2.4. El problema de la mochila

El problema de la mochila

- Se tiene una mochila de capacidad de peso p y una lista de n objetos para colocar en la mochila. Cada objeto i tiene un peso w_i y un valor v_i. Considerando la posibilidad de colocar el mismo objeto varias veces en la mochila, el problema consiste en determinar la forma de colocar los objetos en la mochila sin sobrepasar la capacidad de la mochila colocando el máximo valor posible.

- Los pesos son número enteros.

```
type Peso = Int
```

- Los valores son números reales.

```
type Valor = Float
```

- Los objetos son pares formado por un peso y un valor.

```
type Objeto = (Peso,Valor)
```

- Una solución del problema de la mochila es una lista de objetos.

```
type SolMoch = [Objeto]
```

- Los estados del problema de la mochila son 5–tuplas de la forma `(v,p,l,o,s)` donde

    - `v` es el valor de los objetos colocados,
    - `p` es el peso de los objetos colocados,
    - `l` es el límite de la capacidad de la mochila,
    - `o` es la lista de los objetos colocados (ordenados de forma creciente según sus pesos) y
    - `s` es la solución parcial.

```
type NodoMoch = (Valor,Peso,Peso,[Objeto],SolMoch)
```

- `(sucesoresMoch e)` es la lista de los sucesores del estado `e` en el problema de la mochila.

```
sucesoresMoch :: NodoMoch -> [NodoMoch]
sucesoresMoch (v,p,limite,objetos,solp)
    = [( v+v',
         p+p',
         limite,
         [o | o@(p'',_) <- objetos,(p''>=p')],
         (p',v'):solp )
       | (p',v') <- objetos,
         p+p' <= limite]
```

- (esObjetivoMoch e) se verifica si e es un estado final el problema de la mochila.

```
esObjetivoMoch :: NodoMoch -> Bool
esObjetivoMoch (_,p,limite,((p',_):_),_) =
   p+p'>limite
```

Solución del problema de la mochila por EE

- (buscaEE_Mochila os l) es la solución del problema de la mochila para la lista de objetos os y el límite de capacidad l. Por ejemplo,

```
> buscaEE_Mochila [(2,3),(3,5),(4,6),(5,10)] 8
([(5,10.0),(3,5.0)],15.0)
> buscaEE_Mochila [(2,3),(3,5),(5,6)] 10
([(3,5.0),(3,5.0),(2,3.0),(2,3.0)],16.0)
> buscaEE_Mochila [(2,2.8),(3,4.4),(5,6.1)] 10
([(3,4.4),(3,4.4),(2,2.8),(2,2.8)],14.4)
```

```
buscaEE_Mochila :: [Objeto] -> Peso -> (SolMoch,Valor)
buscaEE_Mochila objetos limite = (sol,v)
    where
      (v,_,_,_,sol) =
          maximum (buscaEE sucesoresMoch
                           esObjetivoMoch
                           (0,0,limite,sort objetos,[]))
```

23.3. Búsqueda por primero el mejor

23.3.1. El patrón de búsqueda por primero el mejor

El patrón de búsqueda por primero el mejor

- (buscaPM s o e) es la lista de soluciones del problema de espacio de estado definido por la función sucesores (s), el objetivo (o) y estado inicial (e), obtenidas buscando por primero el mejor.

```
import ColaDePrioridadConMonticulos

buscaPM :: (Ord nodo) => (nodo -> [nodo]) -> (nodo -> Bool)
                         -> nodo -> [(nodo,Int)]
buscaPM sucesores esFinal x = busca' (inserta x vacia) 0
 where
   busca' c t
    | esVacia c = []
    | esFinal (primero c)
       = ((primero c),t+1):(busca' (resto c)(t+1))
    | otherwise
       = busca' (foldr inserta (resto c) (sucesores x)) (t+1)
         where x = primero c
```

23.3.2. El problema del 8 puzzle por BPM

El problema del 8 puzzle por BPM

- (heur1 t) es la suma de la distancia Manhatan desde la posición de cada objeto del tablero t a su posición en el estado final. Por ejemplo,

```
heur1 inicial8P  ⇝  12
```

```
heur1 :: Tablero  -> Int
heur1 b =
    sum [distancia (b!i) (final8P!i) | i <- [0..8]]
```

- Dos estados se consideran iguales si tienen la misma heurística.

```
instance Eq Tableros
    where Est(t1:_) == Est(t2:_) = heur1 t1 == heur1 t2
```

- Un estado es menor o igual que otro si tiene una heurística menor o igual.

```
instance Ord Tableros where
    Est (t1:_) <= Est (t2:_) = heur1 t1 <= heur1 t2
```

- (buscaPM_8P) es la lista de las soluciones del 8 puzzle por búsqueda primero el mejor.

```
buscaPM_8P = buscaPM sucesores8P
                     esFinal8P
                     (Est [inicial8P])
```

- (nSolucionesPM_8P) es el número de soluciones del 8 puzzle por búsqueda primero el mejor. Por ejemplo,

```
nSolucionesPM_8P  ⇝  43
```

```
nSolucionesPM_8P = length ls
    where (((Est ls),_):_) = buscaPM sucesores8P
                                     esFinal8P
                                     (Est [inicial8P])
```

23.4. Búsqueda en escalada

23.4.1. El patrón de búsqueda en escalada

El patrón de búsqueda en escalada

- (buscaEscalada s o e) es la lista de soluciones del problema de espacio de estado definido por la función sucesores (s), el objetivo (o) y estado inicial (e), obtenidas buscando por escalada.

```
buscaEscalada :: Ord nodo => (nodo -> [nodo])
                 -> (nodo -> Bool) -> nodo -> [nodo]
buscaEscalada sucesores esFinal x =
    busca' (inserta x vacia) where
    busca' c
        | esVacia c           = []
        | esFinal (primero c) = [primero c]
        | otherwise           =
```

```
            busca' (foldr inserta vacia (sucesores x))
            where x = primero c
```

23.4.2. El problema del cambio de monedas por escalada

- El problema del cambio de monedas consiste en determinar cómo conseguir una cantidad usando el menor número de monedas disponibles.
- Las monedas son números enteros.

```
type Moneda = Int
```

- `monedas` es la lista del tipo de monedas disponibles. Se supone que hay un número infinito de monedas de cada tipo.

```
monedas :: [Moneda]
monedas = [1,2,5,10,20,50,100]
```

- Las soluciones son listas de monedas.

```
type Soluciones = [Moneda]
```

- Los estados son pares formados por la cantidad que falta y la lista de monedas usadas.

```
type NodoMonedas = (Int, [Moneda])
```

- `(sucesoresMonedas e)` es la lista de los sucesores del estado `e` en el problema de las monedas. Por ejemplo,

```
ghci> sucesoresMonedas (199,[])
[(198,[1]),(197,[2]),(194,[5]),(189,[10]),
 (179,[20]),(149,[50]),(99,[100])]
```

```
sucesoresMonedas :: NodoMonedas -> [NodoMonedas]
sucesoresMonedas (r,p) =
    [(r-c,c:p) | c <- monedas, r-c >= 0]
```

- `(esFinalMonedas e)` se verifica si `e` es un estado final del problema de las monedas.

```
esFinalMonedas :: NodoMonedas -> Bool
esFinalMonedas (v,_) = v==0
```

- (cambio n) es la solución del problema de las monedas por búsqueda en escalada. Por ejemplo,

```
cambio 199  ⇝  [2,2,5,20,20,50,100]
```

```
cambio :: Int -> Soluciones
cambio n =
    snd (head (buscaEscalada sucesoresMonedas
                             esFinalMonedas
                             (n,[])))
```

23.4.3. El algoritmo de Prim del árbol de expansión mínimo por escalada

- Ejemplo de grafo.

```
g1 :: Grafo Int Int
g1 = creaGrafo True (1,5) [(1,2,12),(1,3,34),(1,5,78),
                           (2,4,55),(2,5,32),
                           (3,4,61),(3,5,44),
                           (4,5,93)]
```

- Una arista esta formada dos nodos junto con su peso.

```
type Arista a b = (a,a,b)
```

- Un nodo (NodoAEM (p,t,r,aem)) está formado por
  - el peso p de la última arista añadida el árbol de expansión mínimo (aem),
  - la lista t de nodos del grafo que están en el aem,
  - la lista r de nodos del grafo que no están en el aem y
  - el aem.

```
type NodoAEM a b = (b,[a],[a],[Arista a b])
```

- (sucesoresAEM g n) es la lista de los sucesores del nodo n en el grafo g. Por ejemplo,

```
ghci> sucesoresAEM g1 (0,[1],[2..5],[])
[(12,[2,1],[3,4,5],[(1,2,12)]),
 (34,[3,1],[2,4,5],[(1,3,34)]),
 (78,[5,1],[2,3,4],[(1,5,78)])]
```

```
sucesoresAEM :: (Ix a,Num b) => (Grafo a b) -> (NodoAEM a b)
                           -> [(NodoAEM a b)]
sucesoresAEM g (_,t,r,aem)
        = [(peso x y g, (y:t), delete y r, (x,y,peso x y g):aem)
           | x <- t , y <- r, aristaEn g (x,y)]
```

- (esFinalAEM n) se verifica si n es un estado final; es decir, si no queda ningún elemento en la lista de nodos sin colocar en el árbol de expansión mínimo.

```
esFinalAEM (_,_,[],_) = True
esFinalAEM _          = False
```

- (prim g) es el árbol de expansión mínimo del grafo g, por el algoritmo de Prim como búsqueda en escalada. Por ejemplo,

```
prim g1 ⇝ [(2,4,55),(1,3,34),(2,5,32),(1,2,12)]
```

```
prim g = sol
    where [(_,_,_,sol)] = buscaEscalada (sucesoresAEM g)
                                        esFinalAEM
                                        (0,[n],ns,[])
          (n:ns) = nodos g
```

Tema 24

Técnicas de diseño ascendente de algoritmos

24.1. Programación dinámica

24.1.1. Introducción a la programación dinámica

Divide y vencerás vs programación dinámica

- Inconveniente de la técnica divide y vencerás: la posibilidad de crear idénticos supbroblemas y repetición del trabajo.
- Idea de la programación dinámica: resolver primero los subproblemas menores, guardar los resultados y usar los resultados de los subproblemas intermedios para resolver los mayores.

Cálculo de Fibonacci por divide y vencerás

- Definición de Fibonacci por divide y vencerás.

```
fib 0 = 0
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

- Cálculo de (fib 4) por divide y vencerás

```
                    fib 4
                  /     \
           +-----+       +--+
           |                |
```

```
            fib 3             fib 2
           /     \           /     \
      fib 2       fib 1 fib 1       fib 0
     /     \
fib 1       fib 0
```

Calcula 2 veces (fib 2) y 3 veces (fib 1) y (fib 0).

Cálculo de Fibonacci por programación dinámica

- Cálculo de (fib 4) por programación dinámica

```
fib 0
 |    fib 1
 |     |
 +-----+=== fib 2
       |     |
       +-----+=== fib 3
             |     |
             +-----+=== fib 4
```

24.1.2. El patrón de la programación dinámica

- Cabecera del módulo:

```
module Dinamica (module Tabla, dinamica)  where
```

- Librerías auxiliares

```
-- Hay que elegir una implementación de TAD Tabla
-- import TablaConFunciones as Tabla
import TablaConListasDeAsociacion as Tabla
-- import TablaConMatrices as Tabla

import Data.Array
```

- El patrón de la programación dinámica

```
dinamica :: Ix i => (Tabla i v -> i -> v) -> (i,i)
                    -> Tabla i v
dinamica calcula cotas = t
    where t = tabla [(i,calcula t i) | i <- range cotas]
```

- `(calcula t i)` es el valor del índice `i` calculado a partir de los anteriores que ya se encuentran en la tabla `t`.
- `cotas` son las cotas de la matriz `t` en la que se almacenan los valores calculados.

24.2. Fibonacci como ejemplo de programación dinámica

24.2.1. Definición de Fibonacci mediante programación dinámica

Definición de Fibonacci mediante programación dinámica

- Importación del patrón de programación dinámica

```
import Dinamica
```

- `(fib n)` es el n-ésimo término de la sucesión de Fibonacci, calculado mediante programación dinámica. Por ejemplo,

```
fib 8  ⇝  21
```

```
fib :: Int -> Int
fib n = valor t n
    where t = dinamica calculaFib (cotasFib n)
```

- `(calculaFib t i)` es el valor de i-ésimo término de la sucesión de Fibonacci calculado mediante la tabla t que contiene los anteriores. Por ejemplo,

```
calculaFib (tabla []) 0                         ⇝ 0
calculaFib (tabla [(0,0),(1,1),(2,1),(3,2)] 4 ⇝ 3
```

Además,

```
ghci> dinamica calculaFib (0,6)
Tbl [(0,0),(1,1),(2,1),(3,2),(4,3),(5,5),(6,8)]
```

```
calculaFib :: Tabla Int Int -> Int -> Int
calculaFib t i
   | i <= 1    = i
   | otherwise = valor t (i-1) + valor t (i-2)
```

- `(cotasFib n)` son las cotas del vector que se necesita para calcular el n-ésimo término de la sucesión de Fibonacci mediante programación dinámica.

```
cotasFib :: Int -> (Int,Int)
cotasFib n = (0,n)
```

Definición de Fibonacci mediante divide y vencerás

- (fibR n) es el n–ésimo término de la sucesión de Fibonacci calculado mediante divide y vencerás.

```
fibR :: Int -> Int
fibR 0 = 0
fibR 1 = 1
fibR n = fibR (n-1) + fibR (n-2)
```

- Comparación:

```
ghci> fib 30
832040
(0.01 secs, 0 bytes)
ghci> fibR 30
832040
(6.46 secs, 222602404 bytes)
```

Definición de Fibonacci mediante evaluación perezosa

- fibs es la lista de los términos de la sucesión de Fibonacci. Por ejemplo,

```
take 10 fibs  ~>  [0,1,1,2,3,5,8,13,21,34]
```

```
fibs :: [Int]
fibs = 0:1:[x+y | (x,y) <- zip fibs (tail fibs)]
```

- (fib' n) es el n-ésimo término de la sucesión de Fibonacci, calculado a partir de fibs. Por ejemplo,

```
fib' 8  ~>  21
```

```
fib' :: Int -> Int
fib' n = fibs!!n
```

- Comparaciones:

```
ghci> fib 30
832040
(0.02 secs, 524808 bytes)
ghci> fib' 30
832040
(0.01 secs, 542384 bytes)
ghci> fibR 30
832040
(6.46 secs, 222602404 bytes)
```

24.3. Producto de cadenas de matrices (PCM)

24.3.1. Descripción del problema PCM

Descripción del problema

- Para multiplicar una matriz de orden $m * p$ y otra de orden $p * n$ se necesitan mnp multiplicaciones de elementos.
- El problema del producto de una cadena de matrices (en inglés, "matrix chain multiplication") consiste en dada una sucesión de matrices encontrar la manera de multiplicarlas usando el menor número de productos de elementos.
- Ejemplo: Dada la sucesión de matrices

$$A(30x1), B(1x40), C(40x10), D(10x25)$$

las productos necesarios en las posibles asociaciones son

$$\begin{array}{lrcrcrcr}
((AB)C)D & 30x1x40 & + & 30x40x10 & + & 30x10x25 & = & 20700 \\
A(B(CD)) & 40x10x25 & + & 1x40x25 & + & 30x1x25 & = & 11750 \\
(AB)(CD) & 30x1x40 & + & 40x10x25 & + & 30x40x25 & = & 41200 \\
A((BC)D) & 1x40x10 & + & 1x10x25 & + & 30x1x25 & = & 1400 \\
(A(BC))D & 1x40x10 & + & 30x1x10 & + & 30x10x25 & = & 8200
\end{array}$$

El algoritmo del PCM

- El PCM correspondiente a la sucesión $d_0, \dots, d_n$ consiste en encontrar la manera de multiplicar una sucesión de matrices $A_1, \dots, A_n$ (tal que el orden de A_i es $d_{i-1} \times d_i$) usando el menor número de productos de elementos.
- Sea $c_{i,j}$ el mínimo número de multiplicaciones necesarias para multiplicar la cadena $A_i, \dots, A_j$ $(1 \leq i \leq j \leq n)$.

- Relación de recurrencia de $c_{i,j}$:
  $c_{i,i} = 0$
  $c_{i,j} = minimo\{c_{i,k} + c_{k+1,j} + d_{i-1}d_kd_j | i \leq k < j\}$
- La solución del problema es $c_{1,n}$.

24.3.2. Solución del PCM mediante programación dinámica

- Importación de librerías auxiliares:

```
import Dinamica
```

- `Cadena` representa el producto de una cadena de matrices. Por ejemplo,

```
P (A 1) (P (A 2) (A 3))  ⇝  (A1*(A2*A3))
P (P (A 1) (A 2)) (A 3)  ⇝  ((A1*A2)*A3)
```

```
data Cadena = A Int
            | P Cadena Cadena

instance Show Cadena where
    show (A x)     = "A" ++ show x
    show (P p1 p2) = concat ["(",show p1,"*",show p2,")"]
```

- Los índices de la matriz de cálculo son de la forma `(i,j)` y sus valores `(v,k)` donde `v` es el mínimo número de multiplicaciones necesarias para multiplicar la cadena $A_i, \ldots, A_j$ y `k` es la posición donde dividir la cadena de forma óptima.

```
type IndicePCM = (Int,Int)
type ValorPCM  = (Int,Int)
```

- `(pcm ds)` es el par formado por el mínimo número de multiplicaciones elementales para multiplicar una sucesión de matrices $A_1, \ldots, A_n$ (tal que el orden de A_i es $d_{i-1} \times d_i$ y $ds = [d_0, \ldots, d_n]$). Por ejemplo,

```
pcm [30,1,40,10,25]  ⇝ (1400,(A1*((A2*A3)*A4)))
```

```
pcm :: [Int] -> (Int, Cadena)
pcm ds = (v, cadena t 1 n)
    where n     = length ds - 1
          t     = dinamica (calculaPCM ds) (cotasPCM n)
          (v,_) = valor t (1,n)
```

- (calculaPCM ds t (i,j)) es el valor del índice (i,j) calculado a partir de la lista ds de dimensiones de las matrices y la tabla t de valores previamente calculados.

```
calculaPCM :: [Int] -> Tabla IndicePCM ValorPCM
              -> IndicePCM -> ValorPCM
calculaPCM ds t (i,j)
    | i == j    = (0,i)
    | otherwise =
         minimum [(fst(valor t (i,k))
                  + fst(valor t (k+1,j))
                  + ds!!(i-1) * ds!!k * ds!!j, k)
                  | k <- [i..j-1]]
```

- (cotasPCM n) son las cotas de los índices para el producto de una cadena de n matrices.

```
cotasPCM :: Int -> (IndicePCM,IndicePCM)
cotasPCM n = ((1,1),(n,n))
```

- (cadena t i j) es la cadena que resultar de agrupar las matrices $A_i, \ldots, A_j$ según los valores de la tabla t.

```
cadena :: Tabla IndicePCM ValorPCM -> Int -> Int -> Cadena
cadena t i j
    | i == j-1  = P (A i) (A j)
    | k == i    = P (A i) (cadena t (i+1) j)
    | k == j-1  = P (cadena t i (j-1)) (A j)
    | otherwise = P (cadena t i (k-1)) (cadena t k j)
    where (_,k) = valor t (i,j)
```

- (pcm' ds) es la lista de los índices y valores usados en el cálculo del mínimo número de multiplicaciones necesarias para multiplicar una sucesión de matrices $A_1, \ldots, A_n$ (tal que el orden de A_i es $d_{i-1} \times d_i$ y $ds = [d_0, \ldots, d_n]$). Por ejemplo,

```
ghci> pcm' [30,1,40,10,25]
[((1,1),(0,1)),((1,2),(1200,1)),((1,3),(700,1)),((1,4),(1400,1)),
 ((2,2),(0,2)),((2,3),(400,2)),((2,4),(650,3)),
 ((3,3),(0,3)),((3,4),(10000,3)),
 ((4,4),(0,4))]
```

```
pcm' :: [Int] -> [((Int, Int), ValorPCM)]
pcm' ds = [((i,j),valor t (i,j)) | i <- [1..n], j <- [i..n]]
```

```
    where n = length ds - 1
          t = dinamica (calculaPCM ds) (cotasPCM n)
```

24.3.3. Solución del PCM mediante divide y vencerás

- (pcmDyV ds) es la solución del PCM correspondiente a ds mediante divide y vencerás. Por ejemplo,

```
pcmDyV [30,1,40,10,25]  ⇝  (1040,(A1*((A2*A3)*A4)))
```

```
pcmDyV :: [Int] -> (Int, Cadena)
pcmDyV ds = cadenaDyV ds 1 n
    where n = length ds - 1
```

- cadenaDyV ds i j) es la solución del PCM correspondiente a $[d_i, \ldots, d_j]$. Por ejemplo,

```
cadenaDyV [30,1,40,10,25] 1 4  ⇝  (1040,(A1*((A2*A3)*A4)))
cadenaDyV [30,1,40,10,25] 2 4  ⇝  (290,((A2*A3)*A4))
```

```
cadenaDyV :: [Int] -> Int -> Int -> (Int, Cadena)
cadenaDyV ds i j
    | i == j    = (0, A i)
    | i == j-1  = (ds!!1*ds!!2, P (A i) (A j))
    | k == i    = (v, P (A i) (subcadena (i+1) j))
    | k == j-1  = (v, P (subcadena i (j-1)) (A j))
    | otherwise = (v, P (subcadena i (k-1)) (subcadena k j))
    where (v,k) = minimum [((valor i k)
                            + (valor (k+1) j)
                            + ds!!(i-1) * ds!!k * ds!!j, k)
                           | k <- [i..j-1]]
          valor p q     = fst (cadenaDyV ds p q)
          subcadena p q = snd (cadenaDyV ds p q)
```

Comparación de las métodos de solucionar el PCM

```
ghci> :set +s

ghci> fst (pcm [1..20])
2658
```

```
(0.80 secs, 39158964 bytes)

ghci> fst (pcmDyV [1..20])
1374
(2871.47 secs, 133619742764 bytes)
```

24.4. Árboles binarios de búsqueda optimales (ABBO)

24.4.1. Descripción del problema de ABBO

Descripción del problema de ABBO

- Para cada clave c_i, sea p_i la probabilidad de acceso a c_i.
- Un árbol binario de búsqueda es optimal (ABBO) si la media del número de comparaciones para todas las claves

  $$a(T) = \Sigma d_i p_i$$

  donde d_i es la distancia de la clave c_i a la raíz (es decir, el número de comparaciones necesarias para llegar a c_i), es mínima.

El algoritmo del ABBO

- Sea $c_{i,j}$ el mínimo valor $a(T)$ cuando el árbol T contiene las claves $c_i, \dots, c_j$.
- Relación de recurrencia para calcular $c_{i,j}$:
  - Si $i > j$, $c_{i,j} = 0$.
  - Si $i = j$, $c_{i,j} = p_i$.
  - Si $i < j$,

    $$c_{i,j} = min_{i \leq k \leq j}((c_{i,k-1} + \sum_{l=i}^{l=k-1} p_l) + (c_{k+1,j} + \sum_{l=k+1}^{l=j} p_l) + p_k)$$
  - El tercer caso puede simplificarse

    $$c_{i,j} = min_{i \leq k \leq j}(c_{i,k-1} + c_{k+1,j}) + \sum_{l=i}^{l=j} p(l)$$

24.4.2. Solución del ABBO mediante programación dinámica

- En la matriz de cálculo del ABBO el valor `(v,k)` correspondiente al índice `(i,j)` indica que `v` es el mínimo valor `a(T)` cuando el árbol `T` contiene las claves $c_i, \ldots, c_j$ y que la división óptima se obtiene dividiendo las claves en dos mediante c_k.

```
type Indice = (Int,Int)
type Valor  = (Float,Int)
```

- `(ABB a)` es el tipo de los árboles binarios de búsqueda sobre `a`.

```
data ABB a = Vacio
           | Nodo a (ABB a) (ABB a)
           deriving Show
```

- `(abbo cs ps)` es el par formado por un ABBO correspondiente a la lista de claves `cs` cuyas correspondientes probabilidades de acceso son los elementos de la lista ps y por su valor. Por ejemplo,

```
ghci> abbo ejProblema
(Nodo 4 (Nodo 1 Vacio
                (Nodo 3 Vacio Vacio))
        (Nodo 10
              (Nodo 8 Vacio Vacio)
              (Nodo 15
                    (Nodo 11 Vacio Vacio)
                    Vacio)),
2.15)
```

- Definición de `abbo`:

```
abbo :: Problema  -> (ABB Int,Float)
abbo pb = (solucion c t (1,n) , fst (valor t (1,n)))
    where (cs,ps) = pb
          n       = length ps
          c       = listArray (1,n) cs
          p       = listArray (1,n) ps
          t       = dinamica (calcula p) (cotas n)
```

- `(calcula p t (i,j))` es el valor del índice `(i,j)` donde `p` es el vector de probabilidades y `t` es la tabla calculada hasta el momento.

```
calcula :: Array Int Float -> Tabla Indice Valor
           -> Indice -> Valor
calcula p t (i,j)
    | i > j      = (0.0,0)
    | i == j     = (p!i,i)
    | otherwise = suma1 (minimum [(fst(valor t (i,k-1))
                                   + fst(valor t (k+1,j)), k)
                                   | k <- [i..j]])
                        (sumaSegmento i j p)
                  where suma1 (x,y) z = (x+z,y)
```

- (sumaSegmento i j p) es la suma de los valores de los elementos del vector p desde la posición i a la j. Por ejemplo,

```
> sumaSegmento 2 4 (array (1,5)
                          [(i,fromIntegral i/2) | i <- [1..5]])
4.5
```

```
sumaSegmento :: Int -> Int -> Array Int Float -> Float
sumaSegmento i j p = sum [p!l | l <- [i..j]]
```

- (cotas n) son las cotas de la matriz revesaria para resolver el problema del árbol de búsqueda minimal óptimo con n claves.

```
cotas :: Int -> ((Int,Int),(Int,Int))
cotas n = ((1,0),(n+1,n))
```

- (solucion cs c (i,j)) es el ABBO correspondiente a las claves c(i),...,c(j) a partir de la tabla de cálculo t.

```
solucion :: Array Int Int -> Tabla Indice Valor
            -> Indice -> ABB Int
solucion cs t (i,j)
        | i > j      = Vacio
        | i == j     = Nodo c Vacio Vacio
        | otherwise = Nodo c (solucion cs t (i,k-1))
                              (solucion cs t (k+1,j))
        where (_,k) = valor t (i,j)
              c     = cs ! k
```

24.5. Caminos mínimos entre todos los pares de nodos de un grafo(CM)

24.5.1. Descripción del problema

- Cálculo de los caminos de coste mínimo entre todos los pares de nodos de un grafo no dirigido.
- Notación:
    - $c_{i,j}$ es el mínimo coste del camino del vértice i al j.
    - $p_{i,j} = \begin{cases} 0, & \text{si } i = j \\ \text{peso del arco entre } i \text{ y } j, & \text{si } i \neq j \text{ y hay arco de } i \text{ a } j \\ \infty, & \text{en otro caso} \end{cases}$
    - $c_{i,j,k}$ es el mínimo coste del camino del vértice i al j, usando los vértices $1, \ldots, k$.
- Relación de recurrencia para calcular $c_{i,j}$:
    - $c_{i,j,0} = p_{i,j}$
    - $c_{i,j,k} = min\{c_{i,j,k-1)}, c_{i,k,k-1} + c_{k,j,k-1}\}$
- El algoritmo se conoce como el algoritmo de Floyd.

24.5.2. Solución del problema de los caminos mínimos (CM)

- Importación de librerías auxiliares:

```
import Dinamica

-- Nota: Elegir una implementación de los grafos.
import GrafoConVectorDeAdyacencia
-- import GrafoConMatrizDeAdyacencia
```

- Ejemplos de grafos para el problema:

```
ej1Grafo :: Grafo Int Int
ej1Grafo = creaGrafo True (1,6)
                     [(i,j,(v!!(i-1))!!(j-1))
                      | i <- [1..6], j <- [1..6]]
```

```
v::[[Int]]
v = [[  0,  4,  1,  6,100,100],
     [  4,  0,  1,100,  5,100],
     [  1,  1,  0,100,  8,  2],
     [  6,100,100,  0,100,  2],
     [100,  5,  8,100,  0,  5],
     [100,100,  2,  2,  5,  0]]
```

- Ejemplos de grafos para el problema:

```
ej2Grafo :: Grafo Int Int
ej2Grafo = creaGrafo True (1,6)
                     [(i,j,(v'!!(i-1))!!(j-1))
                     | i <- [1..6], j <- [1..6]]

v'::[[Int]]
v' =[[  0,  4,100,100,100,  2],
     [  1,  0,  3,  4,100,100],
     [  6,  3,  0,  7,100,100],
     [  6,100,100,  0,  2,100],
     [100,100,100,  5,  0,100],
     [100,100,100,  2,  3,  0]]
```

- En la matriz del cálculo del camino mínimo, los índices son de la forma `(i,j,k)` y los valores de la forma `(v,xs)` representando que el camino mínimo desde el vértice `i` al `j` usando los vértices `1`, ..., `k` tiene un coste `v` y está fomado por los vértices `xs`.

```
type IndiceCM = (Int,Int,Int)
type ValorCM  = (Int,[Int])
```

- `(caminosMinimos g)` es la lista de los caminos mínimos entre todos los nodos del grafo g junto con sus costes. Por ejemplo,

```
ghci> caminosMinimos ej1Grafo
[((1,2),(2,[1,3,2])),  ((1,3),(1,[1,3])),  ((1,4),(5,[1,3,6,4])),
 ((1,5),(7,[1,3,2,5])),((1,6),(3,[1,3,6])),((2,3),(1,[2,3])),
 ((2,4),(5,[2,3,6,4])),((2,5),(5,[2,5])),  ((2,6),(3,[2,3,6])),
 ((3,4),(4,[3,6,4])),  ((3,5),(6,[3,2,5])),((3,6),(2,[3,6])),
 ((4,5),(7,[4,6,5])),  ((4,6),(2,[4,6])),  ((5,6),(5,[5,6]))]
```

```
caminosMinimos :: (Grafo Int Int) -> [((Int,Int), ValorCM)]
caminosMinimos g =
    [((i,j), valor t (i,j,n)) | i <- [1..n], j <- [i+1..n]]
    where n = length (nodos g)
          t = dinamica (calculaCM g) (cotasCM n)
```

- (`calculaCM g t (i,j,k)`) es el valor del camino mínimo desde el vértice `i` al `j` usando los vértices `1`, ..., `k` del grafo `g` y la tabla `t` de los valores anteriores al índice `(i,j,k)`.

```
calculaCM :: (Grafo Int Int) -> Tabla IndiceCM ValorCM
             -> IndiceCM -> ValorCM
calculaCM g t (i,j,k)
  | k==0      = (peso i j g, if i==j then [i] else [i,j])
  | v1<=v2    = (v1,p)
  | otherwise = (v2,p1++p2)
  where (v1,p)   = valor t (i,j,k-1)
        (a,p1)   = valor t (i,k,k-1)
        (b,_:p2) = valor t (k,j,k-1)
        v2 = a+b
```

- (`cotasCM n`) son las cotas de la matriz para resolver el problema de los caminos mínimos en un grafo con `n` nodos.

```
cotasCM :: Int -> ((Int,Int,Int),(Int,Int,Int))
cotasCM n = ((1,1,0),(n,n,n))
```

24.6. Problema del viajante (PV)

24.6.1. Descripción del problema

- Dado un grafo no dirigido con pesos encontrar una camino en el grafo que visite todos los nodos exactamente una vez y cuyo coste sea mínimo.
- Notación:
  - Los vértices del grafo son $1, 2, \ldots, n$.
  - $p_{i,j} = \begin{cases} 0, & \text{si } i = j \\ \text{peso del arco entre } i \text{ y } j, & \text{si } i \neq j \text{ y hay arco de } i \text{ a } j \\ \infty, & \text{en otro caso} \end{cases}$

  - El vértice inicial y final es el n.
  - $c_{i,S}$ es el camino más corto que comienza en i, termina en n y pasa exactamente una vez por cada uno de los vértices del conjunto S.

- Relación de recurrencia de $c_{i,S}$:
  - $c_{i,\emptyset} = p_{i,n}$, si $i \neq n$.
  - $c_{i,S} = min\{p_{i,j} + c_{j,S-\{j\}} : j \in S\}$, si $i \neq n, i \notin S$.
- La solución es $c_{n,\{1,\dots,n-1\}}$.

24.6.2. Solución del problema del viajante (PV)

- Importación de librerías auxiliares

```
import Dinamica

-- Nota: Elegir una implementación de los grafos.
import GrafoConVectorDeAdyacencia
-- import GrafoConMatrizDeAdyacencia
```

- Nota: Para el PV se usará la representación de los de conjuntos de enteros como números enteros que se describe a continuación.
- Los conjuntos se representan por números enteros.

```
type Conj = Int
```

- `(conj2Lista c)` es la lista de los elementos del conjunto `c`. Por ejemplo,

```
conj2Lista 24  ⇝  [3,4]
conj2Lista 30  ⇝  [1,2,3,4]
conj2Lista 22  ⇝  [1,2,4]
```

```
conj2Lista :: Conj -> [Int]
conj2Lista s = c2l s 0
    where
      c2l 0 _              = []
      c2l n i | odd n      = i : c2l (n `div` 2) (i+1)
              | otherwise = c2l (n `div` 2) (i+1)
```

- `maxConj` es el máximo número que puede pertenecer al conjunto. Depende de la implementación de Haskell.

```
maxConj :: Int
maxConj =
   truncate (logBase 2 (fromIntegral maxInt)) - 1
   where maxInt = maxBound::Int
```

- `vacio` es el conjunto vacío.

```
vacio :: Conj
vacio = 0
```

- `(esVacio c)` se verifica si `c` es el conjunto vacío.

```
esVacio :: Conj -> Bool
esVacio n = n==0
```

- `(conjCompleto n)` es el conjunto de los números desde 1 hasta n.

```
conjCompleto :: Int -> Conj
conjCompleto n
   | (n>=0) && (n<=maxConj) = 2^(n+1)-2
   | otherwise = error ("conjCompleto:" ++ show n)
```

- `(inserta x c)` es el conjunto obtenido añadiendo el elemento `x` al conjunto `c`.

```
inserta :: Int -> Conj -> Conj
inserta i s
    | i>=0 && i<=maxConj = d'*e+m
    | otherwise          = error ("inserta:" ++ show i)
    where (d,m) = divMod s e
          e     = 2^i
          d'    = if odd d then d else d+1
```

- `(elimina x c)` es el conjunto obtenido eliminando el elemento `x` del conjunto `c`.

```
elimina :: Int -> Conj -> Conj
elimina i s = d'*e+m
    where (d,m) = divMod s e
          e = 2^i
          d' = if odd d then d-1 else d
```

- Ejemplo de grafo para el problema:

```
   4       5
+----- 2 -----+
|      |1     |
|  1   |   8  |
1----- 3 -----5
|        \2  /
|  6     2\ /5
+----- 4 --6
```

- La definición del grafo anterior es

```
ej1 :: Grafo Int Int
ej1 = creaGrafo True (1,6)
                     [(i,j,(v1!!(i-1))!!(j-1))
                      | i <- [1..6], j <- [1..6]]
v1::[[Int]]
v1 =[[  0,  4,  1,  6,100,100],
     [  4,  0,  1,100,  5,100],
     [  1,  1,  0,100,  8,  2],
     [  6,100,100,  0,100,  2],
     [100,  5,  8,100,  0,  5],
     [100,100,  2,  2,  5,  0]]
```

- Los índices de la matriz de cálculo son de la forma `(i,S)` y sus valores `(v,xs)` donde `xs` es el camino mínimo desde `i` hasta `n` visitando cada vértice de `S` exactamente una vez y `v` es el coste de `xs`.

```
type IndicePV = (Int,Conj)
type ValorPV  = (Int,[Int])
```

- `(viajante g)` es el par `(v,xs)` donde `xs` es el camino de menor coste que pasa exactamente una vez por todos los nodos del grafo `g` empezando en su último nodo y `v` es su coste. Por ejemplo,

```
ghci> viajante ej1
(20,[6,4,1,3,2,5,6])
```

```
viajante :: Grafo Int Int -> (Int,[Int])
viajante g = valor t (n,conjCompleto (n-1))
```

```
    where n = length (nodos g)
          t = dinamica (calculaPV g n) (cotasPV n)
```

- (calculaPV g n t (i,k)) es el valor del camino mínimo en el grafo g desde i hasta n, calculado usando la tabla t, visitando cada nodo del conjunto k exactamente una vez.

```
calculaPV :: Grafo Int Int -> Int -> Tabla IndicePV ValorPV
           -> IndicePV -> ValorPV
calculaPV g n t (i,k)
    | esVacio k = (peso i n g,[i,n])
    | otherwise = minimum [sumaPrim (valor t (j, elimina j k))
                                    (peso i j g)
                           | j <- conj2Lista k]
    where sumaPrim (v,xs) v' = (v+v',i:xs)
```

- (cotasPV n) son las cotas de la matriz de cálculo del problema del viajante en un grafo con n nodos.

```
cotasPV :: Int -> ((Int,Conj),(Int,Conj))
cotasPV n = ((1,vacio),(n,conjCompleto n))
```

Apéndice A

Resumen de funciones predefinidas de Haskell

1. `x + y` es la suma de x e y.
2. `x - y` es la resta de x e y.
3. `x / y` es el cociente de x entre y.
4. `x ^ y` es x elevado a y.
5. `x == y` se verifica si x es igual a y.
6. `x /= y` se verifica si x es distinto de y.
7. `x < y` se verifica si x es menor que y.
8. `x <= y` se verifica si x es menor o igual que y.
9. `x > y` se verifica si x es mayor que y.
10. `x >= y` se verifica si x es mayor o igual que y.
11. `x && y` es la conjunción de x e y.
12. `x || y` es la disyunción de x e y.
13. `x:ys` es la lista obtenida añadiendo x al principio de ys.
14. `xs ++ ys` es la concatenación de xs e ys.
15. `xs !! n` es el elemento n–ésimo de xs.
16. `f . g` es la composición de f y g.
17. `abs x` es el valor absoluto de x.
18. `and xs` es la conjunción de la lista de booleanos xs.
19. `ceiling x` es el menor entero no menor que x.
20. `chr n` es el carácter cuyo código ASCII es n.
21. `concat xss` es la concatenación de la lista de listas xss.
22. `const x y` es x.

23. `curry f` es la versión curryficada de la función `f`.
24. `div x y` es la división entera de `x` entre `y`.
25. `drop n xs` borra los `n` primeros elementos de `xs`.
26. `dropWhile p xs` borra el mayor prefijo de `xs` cuyos elementos satisfacen el predicado `p`.
27. `elem x ys` se verifica si `x` pertenece a `ys`.
28. `even x` se verifica si `x` es par.
29. `filter p xs` es la lista de elementos de la lista `xs` que verifican el predicado `p`.
30. `flip f x y` es `f y x`.
31. `floor x` es el mayor entero no mayor que `x`.
32. `foldl f e xs` pliega `xs` de izquierda a derecha usando el operador `f` y el valor inicial `e`.
33. `foldr f e xs` pliega `xs` de derecha a izquierda usando el operador `f` y el valor inicial `e`.
34. `fromIntegral x` transforma el número entero `x` al tipo numérico correspondiente.
35. `fst p` es el primer elemento del par `p`.
36. `gcd x y` es el máximo común divisor de de `x` e `y`.
37. `head xs` es el primer elemento de la lista `xs`.
38. `init xs` es la lista obtenida eliminando el último elemento de `xs`.
39. `isSpace x` se verifica si `x` es un espacio.
40. `isUpper x` se verifica si `x` está en mayúscula.
41. `isLower x` se verifica si `x` está en minúscula.
42. `isAlpha x` se verifica si `x` es un carácter alfabético.
43. `isDigit x` se verifica si `x` es un dígito.
44. `isAlphaNum x` se verifica si `x` es un carácter alfanumérico.
45. `iterate f x` es la lista `[x, f(x), f(f(x)), ...]`.
46. `last xs` es el último elemento de la lista `xs`.
47. `length xs` es el número de elementos de la lista `xs`.
48. `map f xs` es la lista obtenida aplicado `f` a cada elemento de `xs`.
49. `max x y` es el máximo de `x` e `y`.
50. `maximum xs` es el máximo elemento de la lista `xs`.
51. `min x y` es el mínimo de `x` e `y`.
52. `minimum xs` es el mínimo elemento de la lista `xs`.
53. `mod x y` es el resto de `x` entre `y`.
54. `not x` es la negación lógica del booleano `x`.

55. `noElem x ys` se verifica si `x` no pertenece a `ys`.
56. `null xs` se verifica si `xs` es la lista vacía.
57. `odd x` se verifica si `x` es impar.
58. `or xs` es la disyunción de la lista de booleanos `xs`.
59. `ord c` es el código ASCII del carácter `c`.
60. `product xs` es el producto de la lista de números `xs`.
61. `rem x y` es el resto de `x` entre `y`.
62. `repeat x` es la lista infinita `[x, x, x, ...]`.
63. `replicate n x` es la lista formada por `n` veces el elemento `x`.
64. `reverse xs` es la inversa de la lista `xs`.
65. `round x` es el redondeo de `x` al entero más cercano.
66. `scanr f e xs` es la lista de los resultados de plegar `xs` por la derecha con `f` y `e`.
67. `show x` es la representación de `x` como cadena.
68. `signum x` es 1 si `x` es positivo, 0 si `x` es cero y -1 si `x` es negativo.
69. `snd p` es el segundo elemento del par `p`.
70. `splitAt n xs` es `(take n xs, drop n xs)`.
71. `sqrt x` es la raíz cuadrada de `x`.
72. `sum xs` es la suma de la lista numérica `xs`.
73. `tail xs` es la lista obtenida eliminando el primer elemento de `xs`.
74. `take n xs` es la lista de los `n` primeros elementos de `xs`.
75. `takeWhile p xs` es el mayor prefijo de `xs` cuyos elementos satisfacen el predicado `p`.
76. `uncurry f` es la versión cartesiana de la función `f`.
77. `until p f x` aplica `f` a `x` hasta que se verifique `p`.
78. `zip xs ys` es la lista de pares formado por los correspondientes elementos de `xs` e `ys`.
79. `zipWith f xs ys` se obtiene aplicando `f` a los correspondientes elementos de `xs` e `ys`.

Apéndice B

Método de Pólya para la resolución de problemas

B.1. Método de Pólya para la resolución de problemas matemáticos

Para resolver un problema se necesita:

Paso 1: Entender el problema

- ¿Cuál es la incógnita?, ¿Cuáles son los datos?
- ¿Cuál es la condición? ¿Es la condición suficiente para determinar la incógnita? ¿Es insuficiente? ¿Redundante? ¿Contradictoria?

Paso 2: Configurar un plan

- ¿Te has encontrado con un problema semejante? ¿O has visto el mismo problema planteado en forma ligeramente diferente?
- ¿Conoces algún problema relacionado con éste? ¿Conoces algún teorema que te pueda ser útil? Mira atentamente la incógnita y trata de recordar un problema que sea familiar y que tenga la misma incógnita o una incógnita similar.
- He aquí un problema relacionado al tuyo y que ya has resuelto ya. ¿Puedes utilizarlo? ¿Puedes utilizar su resultado? ¿Puedes emplear su método? ¿Te hace falta introducir algún elemento auxiliar a fin de poder utilizarlo?
- ¿Puedes enunciar al problema de otra forma? ¿Puedes plantearlo en forma diferente nuevamente? Recurre a las definiciones.

- Si no puedes resolver el problema propuesto, trata de resolver primero algún problema similar. ¿Puedes imaginarte un problema análogo un tanto más accesible? ¿Un problema más general? ¿Un problema más particular? ¿Un problema análogo? ¿Puede resolver una parte del problema? Considera sólo una parte de la condición; descarta la otra parte; ¿en qué medida la incógnita queda ahora determinada? ¿En qué forma puede variar? ¿Puedes deducir algún elemento útil de los datos? ¿Puedes pensar en algunos otros datos apropiados para determinar la incógnita? ¿Puedes cambiar la incógnita? ¿Puedes cambiar la incógnita o los datos, o ambos si es necesario, de tal forma que estén más cercanos entre sí?

- ¿Has empleado todos los datos? ¿Has empleado toda la condición? ¿Has considerado todas las nociones esenciales concernientes al problema?

Paso 3: Ejecutar el plan

- Al ejercutar tu plan de la solución, comprueba cada uno de los pasos

- ¿Puedes ver claramente que el paso es correcto? ¿Puedes demostrarlo?

Paso 4: Examinar la solución obtenida

- ¿Puedes verificar el resultado? ¿Puedes el razonamiento?

- ¿Puedes obtener el resultado en forma diferente? ¿Puedes verlo de golpe? ¿Puedes emplear el resultado o el método en algún otro problema?

G. Polya "Cómo plantear y resolver problemas" (Ed. Trillas, 1978) p. 19

B.2. Método de Pólya para resolver problemas de programación

Para resolver un problema se necesita:

Paso 1: Entender el problema

- ¿Cuáles son las *argumentos*? ¿Cuál es el *resultado*? ¿Cuál es *nombre* de la función? ¿Cuál es su *tipo*?

- ¿Cuál es la *especificación* del problema? ¿Puede satisfacerse la especificación? ¿Es insuficiente? ¿Redundante? ¿Contradictoria? ¿Qué restricciones se suponen sobre los argumentos y el resultado?

- ¿Puedes descomponer el problema en partes? Puede ser útil dibujar diagramas con ejemplos de argumentos y resultados.

Paso 2: Diseñar el programa

- ¿Te has encontrado con un problema semejante? ¿O has visto el mismo problema planteado en forma ligeramente diferente?
- ¿Conoces algún problema *relacionado* con éste? ¿Conoces alguna función que te pueda ser útil? Mira atentamente el tipo y trata de recordar un problema que sea familiar y que tenga el mismo tipo o un tipo similar.
- ¿Conoces algún problema familiar con una *especificación* similar?
- He aquí un problema *relacionado* al tuyo y que ya has resuelto. ¿Puedes utilizarlo? ¿Puedes utilizar su resultado? ¿Puedes emplear su método? ¿Te hace falta introducir alguna función auxiliar a fin de poder utilizarlo?
- Si no puedes resolver el problema propuesto, trata de resolver primero algún problema similar. ¿Puedes imaginarte un problema análogo un tanto más *accesible*? ¿Un problema más *general*? ¿Un problema más *particular*? ¿Un problema *análogo*?
- ¿Puede resolver una *parte* del problema? ¿Puedes deducir algún elemento útil de los datos? ¿Puedes pensar en algunos otros datos apropiados para determinar la incógnita? ¿Puedes cambiar la incógnita? ¿Puedes cambiar la incógnita o los datos, o ambos si es necesario, de tal forma que estén más cercanos entre sí?
- ¿Has empleado todos los datos? ¿Has empleado todas las restricciones sobre los datos? ¿Has considerado todas los requisitos de la especificación?

Paso 3: Escribir el programa

- Al escribir el programa, comprueba cada uno de los pasos y funciones auxiliares.
- ¿Puedes ver claramente que cada paso o función auxiliar es correcta?
- Puedes escribir el programa en *etapas*. Piensas en los diferentes *casos* en los que se divide el problema; en particular, piensas en los diferentes casos para los datos. Puedes pensar en el cálculo de los casos independientemente y *unirlos* para obtener el resultado final
- Puedes pensar en la solución del problema descomponiéndolo en problemas con datos más simples y uniendo las soluciones parciales para obtener la solución del problema; esto es, por *recursión*.

- En su diseño se puede usar problemas más generales o más particulares. Escribe las soluciones de estos problemas; ellas puede servir como guía para la solución del problema original, o se pueden usar en su solución.
- ¿Puedes apoyarte en otros problemas que has resuelto? ¿Pueden usarse? ¿Pueden modificarse? ¿Pueden guiar la solución del problema original?

Paso 4: Examinar la solución obtenida

- ¿Puedes comprobar el funcionamiento del programa sobre una colección de argumentos?
- ¿Puedes comprobar propiedades del programa?
- ¿Puedes escribir el programa en una forma diferente?
- ¿Puedes emplear el programa o el método en algún otro programa?

Simon Thompson *How to program it*, basado en G. Polya *Cómo plantear y resolver problemas*.

Bibliografía

[1] Richard Bird: Introducción a la programación funcional con Haskell. (Prentice Hall, 2000).

[2] Antony Davie: *An Introduction to Functional Programming Systems Using Haskell*. (Cambridge University Press, 1992).

[3] Paul Hudak: *The Haskell School of Expression: Learning Functional Programming through Multimedia*. (Cambridge University Press, 2000).

[4] Graham Hutton: Programming in Haskell. (Cambridge University Press, 2007).

[5] Bryan O'Sullivan, Don Stewart y John Goerzen: Real World Haskell. (O'Reilly, 2008).

[6] F. Rabhi y G. Lapalme *Algorithms: A functional programming approach* (Addison–Wesley, 1999).

[7] Blas C. Ruiz, Francisco Gutiérrez, Pablo Guerrero y José E. Gallardo: *Razonando con Haskell*. (Thompson, 2004).

[8] Simon Thompson: *Haskell: The Craft of Functional Programming*, Second Edition. (Addison-Wesley, 1999).